The Report Committee for Abhishek Tondon
Certifies that this is the approved version of the following report:

Performance Impact of Programmer-Inserted Data Prefetches for Irregular Access Patterns with a Case Study of FMM VList Algorithm

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Lizy K John

George Biros

Performance Impact of Programmer-Inserted Data Prefetches for Irregular Access Patterns with a Case Study of FMM VList Algorithm

by

Abhishek Tondon, B. Tech.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2013

Acknowledgements

I would like to acknowledge the cooperation extended by Dhairya Malhotra, M. Tech, Graduate Student at the Institute for Computational Science, Engineering and Mathematics, The University of Texas at Austin, by letting me use the code authored by him as a Case Study for this report. The experiments conducted for this report use the FMM V-list kernel written by Dhairya as the baseline code, in which I insert software prefetches at strategic points to showcase the performance improvements brought by programmer inserted prefetches. An excerpt of the code is included in Appendix B. Additionally, in section 4.1, I rely on the paper lead-authored by Dhairya to present an overview of the FMM algorithm on a part of which the case study is performed.

Abstract

Performance Impact of Programmer-Inserted Data Prefetches for Irregular Access Patterns with a Case Study of FMM VList Algorithm

Abhishek Tondon, M.S.E.

The University of Texas at Austin, 2013

Supervisor: Lizy K John

Abstract: Data Prefetching is a well-known technique to speed up applications wherein hardware prefetchers or compilers speculatively prefetch data into caches closer to the processor to ensure it's readily available when the processor demands it. Since incorrect speculation leads to prefetching useless data which, in turn, results in wasting memory bandwidth and polluting caches, prefetch mechanisms are usually conservative and prefetch on spotting fairly regular access patterns only. This gives the programmer with a knowledge of application, an opportunity to insert fine-grain software prefetches in the code to clinically prefetch the data that is certain to be demanded but whose access pattern is not too obvious for hardware prefetchers or compiler to detect.

In this study, the author demonstrates the performance improvement obtained by such programmer-inserted prefetches with the case study of an FMM (Fast Multipole Method) VList application kernel run with several different configurations. The VList computation requires computing the Hadamard product of matrices. However, the way each node of the octree is stored in the memory, leads to indirect accessing of elements where memory accesses themselves are not sequential but the pointers pointing to those

memory locations are still stored sequentially. Since compilers do not insert prefetches for indirect accesses, and to hardware, the access pattern appears random, programmer-inserted prefetching is the only solution for such a case. The author demonstrates the performance gain obtained by employing different prefetching choices in terms of what all structures in the code to prefetch and which level of cache to prefetch those to and also presents an analysis of the impact of different configuration parameters on performance gain. The author shows that there are several prefetching combinations which always bring performance gain without ever hurting the performance, and also identifies prefetching to L1 cache and prefetching all data structures in question, as the best prefetching recommendation for this application kernel. It is shown that this one combination gets the highest performance gain for most run configurations and an average performance gain of 10.14% across all run configurations.

Table of Contents

1 INTRODUCTION

Processors run faster than memories. And the processors cannot work unless they have the data to work with. The data resides in memory, and therefore programs have loads and stores. Clearly, if an instruction is waiting on a load from memory, even with out-of-order processors, all instructions truly dependent on that, will have to wait as well. The more loads we have in our applications, the more the number of dependent instructions on those loads. Therefore, to have an idea of how severe is the impact of this 'memory wall' problem, we can look at the fraction of loads/stores versus computation operations in the applications of concern.

Table 1, taken from the work by Phansalkar et. al. [1] lists the instruction mix of SPECCPU2006 benchmarks. There also we see that the percentage of loads and stores as a fraction of total instruction approaches 50% in few cases and is above 40% in many of them.

Integer Benchmarks	% Branches	% Loads	% Stores
400.perlbench	23.3%	23.9%	11.5%
401.bzip2	15.3%	26.4%	8.9%
403.gcc	21.9%	25.6%	13.1%
429.mcf	19.2%	30.6%	8.6%
445.gobmk	20.7%	27.9%	14.2%
456.hmmer	8.4%	40.8%	16.2%
458.sjeng	21.4%	21.1%	8.0%
462.libquantum	27.3%	14.4%	5.0%
464.h264ref	7.5%	35.0%	12.1%
471.omnetpp	20.7%	34.2%	17.7%
473.astar	17.1%	26.9%	4.6%
483.xalancbmk	25.7%	32.1%	9.0%

FP Benchmarks	% Branches	% Loads	% Stores
410.bwaves	0.7%	46.5%	8.5%
416.gamess	7.9%	34.6%	9.2%
433.milc	1.5%	37.3%	10.7%
434.zeusmp	4.0%	28.7%	8.1%
435.gromacs	3.4%	29.4%	14.5%
436.cactusADM	0.2%	46.5%	13.2%
437.leslie3d	3.2%	45.4%	10.6%
444.namd	4.9%	23.3%	6.0%
447.dealII	17.2%	34.6%	7.3%
450.soplex	16.4%	38.9%	7.5%
453.povray	14.3%	30.0%	8.8%
454.calculix	4.6%	31.9%	3.1%
459.GemsFDTD	1.5%	45.1%	10.0%
465.tonto	5.9%	34.8%	10.8%
470.lbm	0.9%	26.3%	8.5%
481.wrf	5.7%	30.7%	7.5%
482.sphinx3	10.2%	30.4%	3.0%

Table 1 - SPECCPU2006 benchmarks instruction mix for combined inputs

Figure 1 below from a talk given by John McCalpin [2] on an Industry Perspective on Performance Characterization, presents a scatter plot of the same for most common scientific applications. We see that this fraction is a good ~30%-50% for a good population of applications.

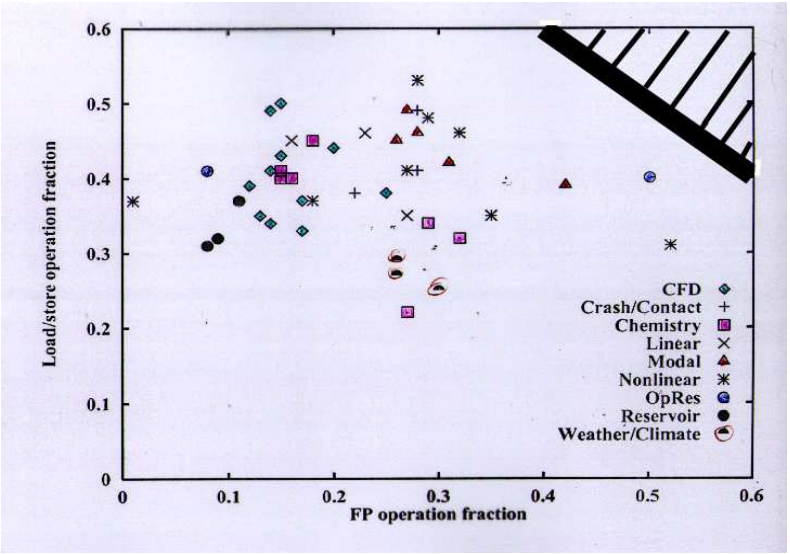

Figure 1 - Fraction of memory instructions in Scientific applications (Source: [2])

The takeaway from the figure and the table above is that there are simply too many memory instructions to take the problem lightly. But then, we have had a solution in the form of caches. Owing to the growing gap between processor and memory speeds, these days, we have up to 3 levels of those. So, should that not solve this problem?

The answer in this regard is a yes and a no. Caches are good when data is being reused. We need some data, we bring it from memory, place a copy in the cache closer to the processor, and work with this copy until an eviction is required, at which time, it is written back to memory, assuming a simplistic single core view. But, for the first time we need this data, it's not in the cache, we have a cold miss and we wait for it to arrive from the memory (or next level cache, as the case might be). We benefit only when the same

data is accessed next time because of temporal locality, or we benefit when there is spatial locality so that the cache line we brought into the cache at the previous miss, also contains some data that we might need next.

While bringing more data than requested finds justification due to the property of spatial locality, another way to look at it is, “bringing data before it is demanded”, which is nothing but a form of data prefetching.

Thus, even without any hardware prefetchers or any compiler or software prefetching, the very act of bringing a cache line which contains some data that is not yet demanded, but, due to spatial locality, is likely to be demanded in near future, can be termed as data prefetching. This underlines how fundamental Data Prefetching is to caching itself.

However, there is a difference too. There has to be some finite granularity at which the data will be brought from lower level memory to a higher level cache. That granularity is the cache line size. Since the cache line does contain the data that was requested, irrespective of whether the extra contiguous data that’s being brought in the cache line, is used or not, that data would have to accompany the data that was actually demanded. This is where data prefetching defers.

In data prefetching, we bring into cache a line that contains none of the data that was demanded. Rather, we bring data closer to the processor on the basis of speculation rooted in access patterns that that data would be demanded in near future. This speculation can be performed either in hardware by hardware prefetchers or in software, either by compiler at higher optimization levels or by programmers by inserting prefetches in the application code.

Both hardware and software approaches have their pros and cons. We’ll look at them in Chapter 2 while discussing briefly the previous work done in both fields.

However, both the hardware prefetchers as well as the compilers can detect fairly regular access patterns only. They tend to be conservative due to the reason that there is a cost involved with prefetching. The cache space is limited and any data that's being prefetched into the cache, would evict some other data. If the speculation was incorrect and prefetched data was not something that would be used in near future, it means polluting the cache and also wasting the precious memory bandwidth. That could potentially hurt the performance instead of helping. That's why these prefetch mechanisms get activated only when the access patterns are fairly regular and result in high confidence level in terms of next accesses.

This is where this study comes into picture. Using the tools described in Chapter 3, this author works on an application kernel, FMM (Fast Multipole Method) VList (described in Section 4.1), which has complex access patterns where hardware prefetchers and compiler inserted prefetches are unlikely to help, and manually insert fine-grain conditional prefetches. The goal is to corroborate the hypothesis that programmer inserted prefetches are indeed likely to bring performance improvement in an application having complex or irregular access patterns. In the course of this study, this author inserts prefetches at strategic points in the FMM VList kernel, and profiles the performance in terms of absolute number of processor cycles taken to execute that.

Hardware prefetchers remain on (default configuration) for all the experiments. Compiler prefeching is turned off and on and does not seem to make any difference in most cases. Several other knobs too are tweaked and the performance gains thus obtained, are profiled. The details of those knobs are covered in Section 4.4 and their impact is discussed through section 5.2 to 5.7. Section 5.8 presents the data and comparison results and a conclusion is drawn on that basis which is presented in Chapter 6. The raw data of these experiments is also included as Appendix C.1 to C.16.

2 BACKGROUND DETAILS

In this chapter, in the first section, a general comparison of hardware and software prefetching is presented, followed by a brief overview of previous work in the field of hardware and software prefetching, the current state of the art and relevant details on programmer inserted prefetches.

2.1 Hardware versus Software prefetching

Broad differences between the two flavors of prefetching are summarized in Table 2 below. Both have their own advantages and disadvantages based on which the preferred one is employed as the use case might require. It's evident from the table that for applications involving irregular access patterns, software prefetching is the solution, that too, programmer inserted ones, as will be clear from section 2.3.

Hardware Prefetchers	**Software Prefetchers**
Regular access patterns	Irregular access patterns
Start-up penalty, more wastage	Doable for smaller ranges with no wastage
Won't cross page boundaries	Page boundaries crossable
Works with existing applications	Need source modification
Sequential and strided prefetch	Compiler inserted, Programmer inserted
Prefetch into fixed cache levels	Prefetch to L1, L2, L3, Non-Temporal Aligned (NTA) buffers

Table 2 - Differences between Hardware and Software prefetching

2.2 Previous Work

Anacker and Wang [4] were the earliest to show the improvements of fetching sequential cache lines beyond just the required word. Their work became the basis of variants like One Block Lookahead [11] and Sequential Prefetching [10] where a block would be prefetched at every access versus at a miss. Another variant, called tagged prefetching [7], involved prefetching a block only if the previously prefetched block was used. A method for measuring the dynamic spatial locality of a program was also proposed by Dahlgran [6] as a means to dynamically control the number of blocks to prefetch ahead. Later called, Adaptive Sequential Prefetching, this scheme was conservatively added to Intel's NetBurst Microarchitecture [8] for prefetching to Unified L2 cache.

Chen and Baer [5] went a step ahead and proposed Stride prefetching combined with the adaptive property of Dahlgran's design. A 2004 survey by Perez. et. Al [9] showed that Stride Prefetching was superior to other mechanisms in terms of performance and power both. Subsequently, this was added to Intel's core microarchitecture.

With regard to software prefetching, Callahan et. al. [13] were the first to propose using a non-blocking prefetch instruction in 1991, and they were able to show >99% hit ratio for demand requests with prefetching on, for their set of benchmarks. Subsequently, an Integrated scheme for Hardware/Software Data prefetching was proposed [14] by Edward Gornish of University of Illinois in 1994 for Shared Memory Multiprocessors. They showed that an Integrated scheme performed better than plain hardware prefetching because of its ability to handle more complex patterns and better than software only approach as it required fewer instructions.

Other prominent related work in this domain includes a study of the interaction of Software prefetching with ILP processors in shared memory systems by Parthasarthy et.al. [15] in 1997 and one studying its efficacy for future memory systems by Badawy et. al. [16] in 2004, among others.

2.3 Current State of the art

Present day Intel processors have more than one type of hardware prefetchers. Intel's website says that the Core i7 and Xeon 5500 series processors have prefetchers that can prefetch into either L2 only or both L1 and L2. They employ more than one algorithms too, which include simpler ones such as fetching 2 cache lines instead of one, when a demand is made, and more sophisticated ones that involve monitoring the access patterns of a cache.

With regard to software prefetching, modern compilers come equipped with the capability to insert prefetches on their own. Compiler prefetching is turned on by default for Optimization level O2 and above for Intel C Compiler for the Knights' Corner platform being used in this experiment. The architecture supports several instructions. Vprefetch1 instruction brings a 64 byte cache line into L2 cache and vprefetch0 pulls it into L1 cache. Their corresponding variations vprefetchet1 and vprefetchet0 do the same while additionally marking the prefetched line as exclusive [12].

The compiler can issue prefetches for all regular memory accesses inside loop [12]. In the process of determining prefetch distances, the compiler takes into account the latency of accessing the target level in memory hierarchy and the expected total latency of the loop to complete its one iteration. Compiler can also generate initial value prefetches before entering a loop to help even the first few iterations. However, compiler does not issue any prefetching for indirect accesses of the form a[b[i]]. Such cases need

to be handled by programmer inserted manual prefetches, which is what is exploited in this work.

To avoid interfering with programmer inserted prefetches, the compiler prefetching can be disabled by using the compile-time switch **-no-opt-prefetch**.

2.4 Programmer Inserted Prefetches

Programmer can insert prefetches using the _mm_prefetch() compiler intrinsic. This takes 2 arguments, first of which is a character pointer pointing to the address to start the prefetch from. The second argument can take one of the 4 following values.

_MM_HINT_T0	Prefetch into L0 cache
_MM_HINT_T1	Prefetch into L1 cache
_MM_HINT_T2	Prefetch into L2 cache
_MM_HINT_NTA	Prefetch into Non-temporal aligned buffers

Table 3 - Prefetch destination for Programmer Inserted Prefetching

Since the caches are inclusive, prefetching to a higher level of cache essentially means prefetching to all levels below it. Prefetching into Non-temporal aligned buffers is useful when the prefetched data is not likely to exhibit temporal locality and therefore, it's not desired to occupy cache space for that data. The header file mmintrin.h needs to be included to use this intrinsic.

3 MEASUREMENT INFRASTRUCTURE

Since this research is about measuring performance gain, it becomes critical to select the right tool(s) which provide(s) trustworthy results. Hardware Performance Counter tools come to mind as natural choice. The tools considered for this research included Intel VTune Amplifier XE 2013 and PAPI (Performance Application Programming Interface) which are discussed in the sections 3.1 and 3.2. The last section of the chapter covers information about validating the chosen tool.

3.1 Intel VTune

The foremost reason behind picking Intel VTune was the availability of a vast set of processor events that are directly measurable with this tool. Coming from the processor manufacturer Intel itself, there was little to doubt the accuracy of the numbers provided by the tool. Moreover, the tool did not require changes in the source and just launching the tool and running the application with command line arguments or with GUI, let's one obtain the events one wants.

However, in the case of this study, this author found severe inconsistencies in the data obtained using VTune. The numbers were wildly fluctuating, often to the order of 30% or more. Clearly, such a high level of variability could not be tolerated.

There were a couple of reasons that this author suspected, were causing this behavior. The first one is inherent to VTune which is that it's based on sampling. It does not keep an accurate count but samples the counters at regular intervals. The sampling interval is chosen by the tool itself using its inbuilt algorithms and the number of events that are being counted, although that can be overridden.

As a first measure, reducing the number of events to make way for finer sampling interval, did not seem to make any difference. The variations remained as wide.

As second measure, this author turned off power cycling, which were suspected to be another reason behind wild variations as it could be causing error in multiplication factors if processor changes its operating frequency in the middle of the run. Unfortunately, even this change did not seem to help VTune numbers stabilize.

Consequently, this author had to abandon the tool and discard the datasets obtained with VTune.

3.2 PAPI

After having an unsatisfactory experience with VTune, this author decided to switch to PAPI [3]. PAPI, Performance Programming Application Interface, specifies a standard API for accessing hardware counters available on the processor.

Evidently, PAPI requires changes in the source of the application. However, it also easily facilitates measuring counters for a specific part of the application. This was a requirement for this research the the FMM VList kernel that this research considers as a Case Study for Programmer Inserted prefetches, is only a part of the complete FMM algorithm.

In this research, the low level interface of PAPI was used. The low level interface allows dealing with hardware events by creating EventSets. Low level functions such as PAPI_add_event, PAPI_start and PAPI_stop are called in addition to PAPI_create_eventset. The code added for PAPI profiling is only a few lines and is included in Appendix A.

We measure the following preset events among others, PAPI_L1_DCM, PAPI_L2_DCM, PAPI_L3_TCM, PAPI_TOT_CYC, PAPI_LD_INS, PAPI_TOT_INS for one thread which, for our configurations, will cover $1/16^{th}$ of the complete task.

Table 4 lists these events and its meanings.

PAPI Preset Event	Meaning of the event (all events per thread)
PAPI_L1_DCM	L1 D-cache misses
PAPI_L2_DCM	L2 D-cache misses
PAPI_L3_TCM	L3 Total cache misses
PAPI_LD_INS	Load instructions count
PAPI_TOT_INS	Total instructions count
PAPI_TOT_CYC	Total cycles taken
PAPI_DP_OPS	Floating point operations

Table 4 - PAPI Event descriptions

Since there are only 5 hardware counters available on the processor, two runs are required to get one set of all these events. Unlike VTune, PAPI numbers show very small variations, of about 2%. To be accurate, 10 runs are made for each run configuration to obtain 5 sets of all event values. Trimmed averages of each such groups are computed and those raw datasets for all run configurations are reported in Appendices C.1 to C.16.

3.3 SPEC Validation of PAPI

As an additional check on the veracity of the numbers reported by PAPI, an analysis of select SPEC CPU2006 benchmarks was undertaken. 15 Integer and Floating point SPEC benchmarks were picked (the ones with code in C or C++) and PAPI calls were inserted in them. The event set for this test comprised PAPI_BR_INS (branch instruction count) and PAPI_SR_INS (store instruction count) in addition to PAPI_LOAD_INS, and PAPI_TOT_INS so as to calculate Instruction mix. The representative reference input sets for those benchmarks which had multiple input sets,

were chosen based on [1]. The instruction mix of these benchmarks are plotted in Figure 2. Figure 3 plots the instruction mix for the same benchmarks as reported in [1] which we use as reference. It is evident that our instruction mix of Figure 2 is visibly close to the reference mix of Figure 3. The slight differences can be attributed to compiler optimizations which have taken place over the duration since [1] was written and also to the fact that for benchmarks with multiple inputs, our instruction mix relies on the representative input sets, which, therefore, results in close but not identical instruction mix.

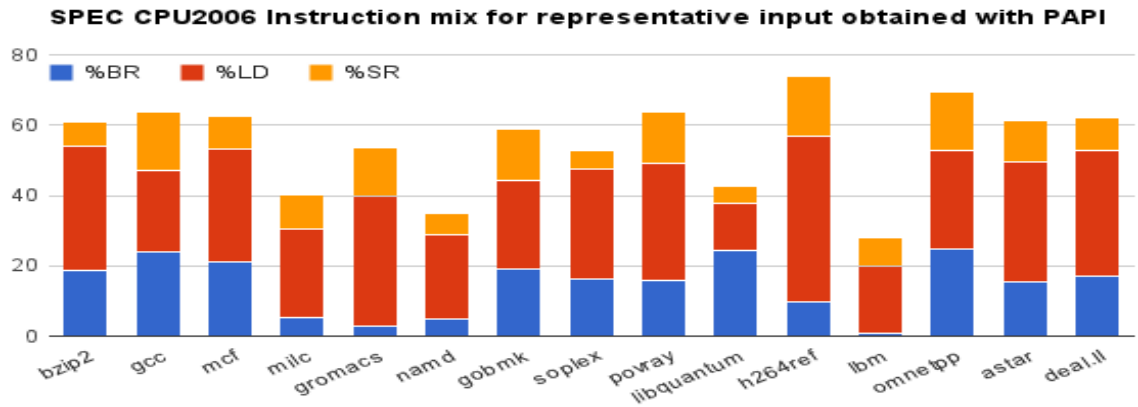

Figure 2 - SPEC CPU2006 Instruction mix with PAPI

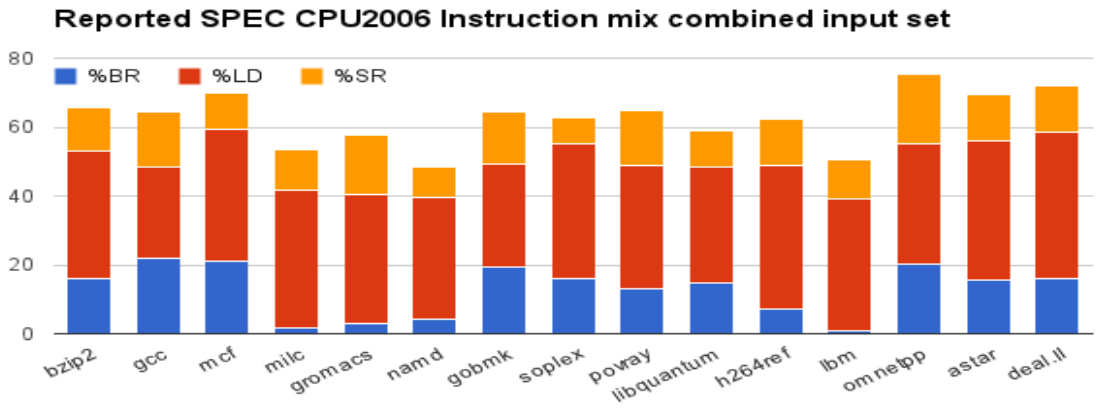

Figure 3 - SPEC CPU2006 Instruction mix reference as reported in [1]

4 CASE STUDY DETAILS

In this chapter, an introduction of the FMM algorithm is presented in section 4.1 with an emphasis of VList computation, which is that step of the algorithm, which forms the focus of this study. In section 4.2, the prefetching scenario is explained highlighting how there is a case for programmer inserted prefetches in this application kernel. The specification of the system on which all experiments are run, is covered in section 4.3 and different experiment configurations are discussed in section 4.4.

4.1 Overview of FMM

FMM, or Fast Multipole Method, is an algorithm for computing volume potentials used to construct spatially adaptive solvers for Poisson and Stokes' equations. Computation domain is partitioned hierarchically by using an Octree structure. Given a set of M point sources with individual charge strengths and N target points, FMM calculates the total potential at each target point. This is essentially an N-body problem which involves pairwise interactions among system of N particles. The inherent complexity for such a problem would be $O(N^2)$ but FMM reduces that by using approximations for far interactions. A Far interaction is approximated by computing multipole expansion, multipole-to-local translation, and evaluating local expansion. The following paragraphs in this section, which draw an explanation from Malhotra et. al's paper [17], explain the concept.

Both the multipole expansion as well as the local expansion for a node are represented by sets of source points on an equivalent surface around the node such that it produces the same potential at the points on a check surface as the original source points. To distinguish between the two, upward-equivalent and upward check surfaces are used for multipole expansion and downward equivalent and downward check are used for

local expansion. For a leaf node, potential at the check surface is computed due to source points, while for non-leaf nodes, the check surface potential from points on the equivalent surface of children is computed followed by computing the equivalent source intensities for leaf nodes.

The FMM algorithm involves an upward pass followed by a downward pass. In the upward pass, the octree is traversed postorder and multipole expansion of each node is computed. For the leaf nodes, it's computed directly from underlying source distribution, and for the non-leaf nodes, it's computed from the multipole expansion of its children. In the downward pass, a preorder traversal is performed, and local expansion are computed for each node. The local expansion of the parent node and the multipole expansions of those nodes which are at the same level as the node in question, and well-separated from it, but not well-separated from its parent node, are added to that. For leaf nodes, the final potential at target points is computed by first computing direct interactions from sources in adjacent leaf nodes and then adding contribution from local expansion.

For non-uniform distribution of points, which translates to adaptive octrees, the interactions are complex. Based on the type of interactions which in turn depend on the relative positions of nodes, all interactions can be classified into 4 lists - U, V, W and X lists. The U-list contains near interactions of the leaf nodes with itself or with an adjacent leaf node. U-list interactions are not approximated and are always computed accurately. The V-list includes contributions to the local expansion of a node from source nodes which are at the same depth in octree, lie completely outside the downward equivalent surface of the node and are within the equivalent surface of its parent node. The W-list includes source nodes that are at a finer level than the target node and for which the target node lies outside the upward check surface of their source octant, but which do not lie

outside the downward equivalent surface of the target node. For such interactions, multipole expansion of source node is used to evaluate the potential directly at target points. The last list, X-list includes source nodes which are leaf nodes, and are at a coarser level than the target node, and for which the target node is not outside its upward check surface but the source nodes are outside the downward-equivalent surface of the target node, allowing the local expansion to be constructed. For such interactions, the downward check potential at the target node is evaluated directly from the points in the source node.

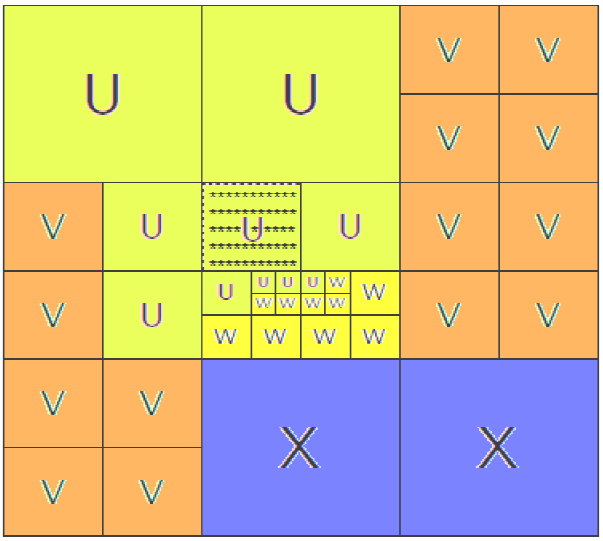

Figure 4 - Interactions in a Quad-tree

It is difficult to graphically explain this for Octrees as the problem requires 3-dimensions, however, the same can be explained for quad-trees in 2-dimensions. Figure 4, drawn from [17], marks the nodes that'll go in different list w.r.t to the central node which is marked with stars, based on their positioning and level of granularity.

In this research, entire focus is on V-list computation step of the algorithm. Complex Hadamard product computation between vectors in Fourier space is needed for V-list computation. There are Interleaved source and target vectors for sibling octants. 8 elements are loaded from source and target vectors, one for each sibling, and then, all interactions are computed among those as matrix-matrix multiplications. The same is repeated for next octet. That is the kernel whose performance is targeted for optimization

by custom inserting prefetches. The arrays and pointers of interest in this kernel and their access patterns are discussed in the next section.

4.2 Prefetchability scenario

The V-list computation step involves the calculation of Hadamard product, which is element by element product for two matrices. However, because of the nature of the algorithm, the addressing is indirect. A simplified view of the same is shown in Figure 5.

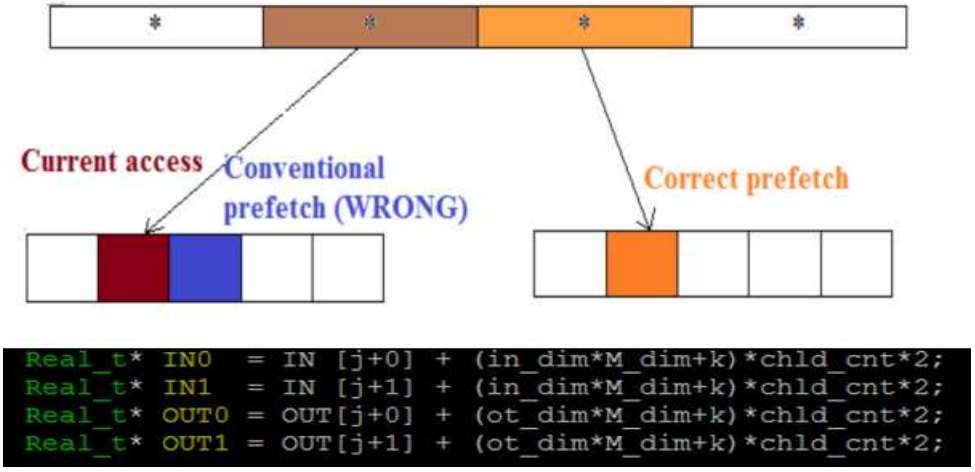

Figure 5 - Simplified view of access patterns

As it's clear from the figure, an array of pointers stores a set of pointers which point to the some elements in an array. In the context of the attached code snapshot, IN and OUT are pointers. The relevant excerpt of the code is available in Appendix B. From the code in Figure 5, it's clear how IN0, IN1 and OUT0, OUT1 point to some element within IN and OUT which further point to some addresses. Since this assignment takes place inside a loop, whose counter is j, each time j changes, IN0, IN1, OUT0, OUT1 end up pointing to a new location. Since the subsequent code performs memory and multiplication operations on IN0, IN1, OUT0 and OUT1, those are the ones which

require prefetching. However, those are neither sequential, nor strided, so hardware prefetchers cannot work on those. Since those are indirect accesses, compiler prefetching won't work on those either. In fact, an attempt to sequentially prefetch would only hurt the performance as in the next iteration of the inner loop, the consecutive next location of the IN0, IN1 or OUT0, OUT1 will not be accessed at all. As the Figure 5 depicts, the locations which are pointed by the next location in IN and OUT, will be accessed.

Therefore, programmer inserted prefetches are the only way to perform prefetching at all. Also, because this all is inside nested loops and j values keep getting reset at each iteration of the outer loop, it becomes important to take care of bound violations. Therefore, the prefetches need to be inserted conditionally and the prefetches need to be performed on IN and OUT instead of IN0, IN1, OUT0, OUT1, as those are the ones which move sequentially, at least within the inner loop iteration.

Other than IN and OUT, there is another pointer, M, which exhibits similar access pattern, albeit at an outer loop, thereby becoming a candidate for prefetch. This gives us several choices in terms of which all arrays to prefetch, and which all levels of cache to prefetch it to. Additionally, there is choice about in how much advance the prefetches should be made. For that, a cursory inspection of the code makes it clear that prefetching for the next iteration is the best. Reason being, there are lots of operations that take place within the inner most loop, which are sufficient to hide the latency of a prefetch. It is also worth mentioning that the code uses Intel AVX extensions and operates on a chunk of 128 byte data at a time. Since the size of a cache line is only 64 bytes, it becomes imperative to prefetch 2 cache lines each for IN0, IN1 and OUT0, OUT1, and M, if at all prefetching is being done for those. These configuration choices and several others are discussed and tabulated in section 4.4. A brief specification of the Stampede machine on which the experiments are conducted, is presented next.

4.3 System specification

All the experiments are carried out on the Stampede Supercomputer of the Texas Advanced Computing Center (TACC). Each node of Stampede consists of 2 Xeon E5-2680 processors and a Xeon Phi SE 10P Coprocessor (referred as MIC). Each Xeon processor has 8 cores running at the frequency of 2.7 GHz. There is 32 KB of L1I Cache and L1D cache per core, and 256 KB of unified L2 Cache per core. There is 20 MB of shared L3 cache per processor (8 cores). Thus, each node has 16 cores and 32 GB memory per node.

4.4 Experiment configurations

In addition to the choice of which all locations to prefetch and which cache levels to prefetch those to, there are several other configuration knobs that need attention.

Since this is a multi-threaded code, there is parallelism available in terms of MPI tasks and OpenMP threads. Since this kernel is a part of a complete algorithm, and cannot be made to run standalone, and the fact that there is only OpenMP parallelism within the VList kernel but number of MPI tasks determine how many of those kernels will run in parallel, also make the number of MPI tasks and OpenMP threads a variable to sweep. Since the algorithm has differences in terms of uniform versus non-uniform distribution, that's yet another variable. Also, while compiler prefetching apparently should not kick in within the kernel of concern, it can certainly affect other portions of the overall algorithm and the cache utilization therefore can affect the kernel too. So, keeping Compiler prefetching on versus keeping it off, is another configuration setting. Finally, the platform where this code is run, also has a coprocessor and parts of the complete algorithm (though not the VList kernel itself) can be offloaded to the coprocessor. That is another knob which can affect the impact of inserted prefetches. Therefore, all these

knobs are varied and the impact of inserted prefetches in profiled across the design space. Table 5(a) below summarizes all these configuration choices that're exercised in this experiment.

What to prefetch	No prefetch	Prefetch IN	Prefetch IN, M arrays	Prefetch IN, OUT arrays	Prefetch IN, OUT, M arrays
Where to prefetch	To L1	To L2	To L3	To NTA	
Distribution	Uniform		Non-uniform		
Parallel threads setting	1 MPI task, 16 OMP threads/task	4 MPI tasks, 4 OMP threads/task	16 MPI tasks, 1 OMP thread/task		
Coprocessor offloading	OFF		ON		
Compiler prefetching	OFF		ON		

Table 5(a) - Experiment Configurations

Some of these combinations are not allowed. For example, "What to prefetch" and "Where to prefetch" together result in 17 different prefetching combinations, 4 "What to prefetch" times 4 "Where to prefetch" and 1 "No prefetch". Note that the IN array is always prefetched if unless it's a no prefetch case. That's because the code cannot proceed until IN0 and IN1 are loaded. Also, with Coprocessor offloading ON, only one parallel thread setting, involving 1 MPI tasks and 16 OMP threads, is used. That's because of the way the overall code is structured, and each task needs parts of it offloaded to coprocessor. Thus, it results in 16 different run configurations. Those run configurations are listed in Table 5(b) below. That, multiplied with 17 prefetching combinations, i.e. 272 sets of values, multiplied with 10 runs made for each set.

Several scripts were written to make required code changes to build the binaries and launch the batch run scripts with required settings, generate the performance logs. The results are presented and analyzed in the next chapter.

Sr. No.	Configuration details	Section 5.8 Table/Fig.
1.	No coprocessor offloading; 1 MPI task,16 OMP threads/task; Uniform distribution; Compiler prefetching off	6
2.	No coprocessor offloading; 1 MPI task,16 OMP threads/task; Uniform distribution; Compiler prefetching on	7
3.	No coprocessor offloading; 1 MPI task,16 OMP threads/task; Non-uniform distribution; Compiler prefetching off	8
4.	No coprocessor offloading; 1 MPI task, 16 OMP threads/task; Non-uniform distribution; Compiler prefetching on	9
5.	No coprocessor offloading; 4 MPI tasks, 4 OMP threads/task; Uniform distribution; Compiler prefetching off	10
6.	No coprocessor offloading; 4 MPI tasks, 4 OMP threads/task; Uniform distribution; Compiler prefetching on	11
7.	No coprocessor offloading; 4 MPI tasks, 4 OMP threads/task; Non-uniform distribution; Compiler prefetching off	12
8.	No coprocessor offloading; 4 MPI tasks, 4 OMP threads/task; Non-uniform distribution; Compiler prefetching on	13
9.	No coprocessor offloading; 16 MPI tasks,1 OMP threads/task; Uniform distribution; Compiler prefetching off	14
10.	No coprocessor offloading; 16 MPI tasks, 1 OMP threads/task; Uniform distribution; Compiler prefetching on	15
11.	No coprocessor offloading; 16 MPI tasks, 1 OMP threads/task; Non-uniform distribution; Compiler prefetching off	16
12.	No coprocessor offloading; 16 MPI tasks, 1 OMP threads/task; Non-uniform distribution; Compiler prefetching on	17
13.	Coprocessor offloading on; 1 MPI task,16 OMP threads/task; Uniform distribution; Compiler prefetching off	18
14.	Coprocessor offloading on; 1 MPI task,16 OMP threads/task; Uniform distribution; Compiler prefetching on	19
15.	Coprocessor offloading on; 1 MPI task,16 OMP threads/task; Non-uniform distribution; Compiler prefetching off	20
16.	Coprocessor offloading on; 1 MPI task,16 OMP threads/task; Non-uniform distribution; Compiler prefetching on	21

Table 5(b) - Run configurations and corresponding tables/figures in section 5.8

5 RESULTS AND ANALYSIS

The 272 sets of values obtained as trimmed averages from 2720 runs are organized as sets of 17 prefetching options under 16 run configurations. All experiments were run for 150000 points in the N-body FMM problem. The raw datasets of trimmed averages are included in Appendix C.1 to C.16. Section 5.8 in this chapter includes the cycle count as the measure of performance and Tables 6 to 21 present those numbers. Absolute number of cycles are plotted for all 17 prefetching options for each of the 16 run configurations in Figure 6(a) to 21(a). The percentage gain/loss for each of the 16 prefetch combination against the 'no prefetch' baseline is plotted for each run configuration as well, in Figure 6(b) to 21(b). The broader results are summarized and analyzed in section 5.1. Sections 5.2 to 5.7 analyze the impact of each configuration or prefetching choice.

5.1 Broader results

Looking at the graphs 6(b) to 21(b), it is evident that in programmer inserted prefetches in general bring significant gains to the performance as manifested in the reduction in absolute number of processor cycles recorded in corresponding tables. There are few run configurations for which some prefetch combinations end up hurting performance, but those are those combinations which are never among the best performers for any run configurations. That is to say, the best prefetching choices never hurt the performance across any run configurations, and consistently bring significant gains across all.

In terms of prefetch destination, prefetching the data to either L1 cache or to NTA buffers brings the most gains. In fact, prefetching to L1 or prefetching to NTA buffers brings nearly the same performance gains for most run configurations which leads the

author to believe that NTA buffers have the same access latency as the L1 cache. Strictly counting, prefetching to L1 cache is found to be the best for 10 out of 16 run configurations, while prefetching to NTA buffers is found to be the best for 5. There's an outlier in that regard for which prefetching to L2 yields the most benefit followed by prefetching to NTA buffers or L1 cache.

In terms of prefetch choices, prefetching all 3 arrays, IN, OUT and M turns out to be the best decision for 12 out of 16 run configurations. There are only 4 configurations for which not prefetching M array proves slightly better than prefetching M. Since M array is prefetched in an outer loop, there are fewer prefetches for it, and the impact is, either way, not substantial.

In general, it can be recommended that the programmer, for this application kernel, should insert prefetches on all 3 structures, IN, OUT and M, and should prefetch those into L1 cache. This, as per the results of this study, would ensure highest percentage performance gain for 8 out of 16 run configurations and second highest percentage performance gain for 4 out of the remaining. This combination of what-to-prefetch and where-to-prefetch performs better than any other combination across different run configurations. On an average, it provides 10.14% of performance gain across all run configurations which is only marginally lower than the average of the individual best prefetching combinations for each run configuration, that comes out to be 10.86%.

The Appendices C.1 to C.16 contain other useful data such as the absolute number of loads, total instructions, and cache misses at different levels. An important thing to note in this regard is that the absolute number of load instructions and consequently total instructions increases as more data is prefetched. The Xeon processor events to which PAPI cache miss preset events map to, are the ones which include both demand misses

and prefetch misses. The same is reflected in increased number of absolute cache misses. Unfortunately, while Intel VTune offered events to separately count demand and prefetch misses, the same is not included in PAPI preset events which count both separately. Since unlike demand misses, prefetch misses do not cause the processor to keep waiting for the data, those prefetch misses do not harm the performance. On the contrary, prefetching ensures data is available for the next iteration at the right time, and the same is reflected in the reduction observed in the number of cycles required with prefetching versus no prefetching.

5.2 Impact of prefetch choices

Other than no prefetching, there are 4 prefetching choices that're studied and profiled in these experiments as listed in Table 5(a). A cursory look at the graphs 6 (b) to 21 (b) reveals that for 13 out of 16 run configurations, the best performance comes when all these 3 are being prefetched. In many cases, prefetching M or not prefetching it makes less difference than not prefetching OUT, and the reason lies in the fact that M is at a much outer loop in the nesting order, as compared to IN and OUT and therefore, undergoes far fewer prefetches. There are also 3 configurations where not prefetching M while still prefetching IN and OUT arrays proves slightly better than prefetching M. A possible reason for that could be destructive interference between the prefetches. There is no economical way to check if two prefetches are interfering in mutually destructive manner on an access by access basis, as even for a fixed configuration, there could be sets of accesses that could interfere constructively, and some other sets destructively. Repeated empirical runs are the only way to ascertain that, because all the address layout changes with so many factors including parallelization settings, type of distribution etc.

Since the idea behind tweaking all the prefetching choices and destination essentially was to come up with the best and consistent combination across all run configurations, it can be safely concluded, that prefetching all 3 locations, IN, OUT and M is the preferred choice.

5.3 Impact of prefetch destination

In terms of prefetch destination, again looking at Figure 6(b) to 21(b) shows that the choice is between prefetching to either L1 cache or to the non-temporal aligned buffers. Prefetching to L2 or L3 yields less performance gains across all runs. Between prefetching to L1 and prefetching to NTA, in 10 out of 16 run configurations, prefetching to L1 gives the best performance along with corresponding prefetch data choice. Among the remaining, 5 have prefetching to NTA performing better but except one of them, the difference is marginal. It is only intuitive that prefetching the data closest to the processor would result in minimal latency and hence maximal performance gain as long as the prefetching choices are correct. Therefore, these results too make sense. Prefetching to NTA buffers for some configurations proves slightly better than prefetching to L1 cache as it would utilize L1 cache's limited space better. However, an important thing to keep in mind here is the fact that while the data brought into IN and OUT arrays is used only for one iteration, the data prefetched into M array is used for several iterations, and therefore, prefetching M array to NTA buffers contributes to lower gains. Again, since the objective is to find the general best prefetching combination across all runs which never hurts performance and provides consistent performance gain, it can be safely said that prefetching to L1 cache is the best bet.

5.4 Impact of Uniform/Non-uniform distribution

Since there is significant difference in the way interactions behave in uniform versus non-uniform distribution of points, the vast difference between baseline performance is understandable. The distribution simply affects the net amount of work to be done and that reflects in the absolute number of cycles. That is inherent in the algorithm and has nothing to do with prefetches.

However, the difference in the relative performance gain due to prefetches should not be significant, as prefetches still work the same way. In that regard, the general trend is that the performance gain observed for uniform distribution is slightly but consistently higher than that observed for non-uniform distribution. An intuitive explanation for that could lie in the count of inner loops. Since the prefetches inserted are conditional, to avoid bound violations, no prefetching is done in the last iteration of the inner loop. Consequently, during the next iteration of the outer loop, the data needed for the first iteration of the inner loop is not prefetched. Since a non-uniform distribution results in inner loops having fewer iterations, that would decrease the impact of prefetching as there will be more prefetch skips per total number of prefetches. That appears to be the reason behind the prefetch performance gain for non-uniform case for best performing combinations a notch lower than the uniform distribution.

For 1 MPI, 16 OMP run configurations without coprocessor offloading, the non-uniform case also exhibits negative gains for those prefetching choices which were low performing for uniform distribution, i.e. Cases when OUT is not prefetched. While a general lower value is explained in the previous paragraph, not prefetching OUT array would further wash away any gains because of more frequently skipping prefetches and a constant branch check overhead. This effect is more pronounced when 16 openMP threads because of the way an outer loop work is divided among openMP threads. Higher

number of threads would mean lesser work per thread and that would make the overhead more pronounced resulting in performance hurt for the unpreferred prefetching choices.

In a nutshell, since the objective is to deal with best and consistent prefetching choices, it can be said that this prefetching methodology performs slightly better for uniform distribution as compared to non-uniform one.

5.5 Impact of MPI/OpenMP configuration

The number of MPI tasks and OpenMP threads per task is a vital choice. This complete affects what addresses a thread will get to work upon and consequently affects both, the baseline performance as well as the prefetching gains.

One clear trend is that MPI parallelism results in better performance as compared to equivalent openMP parallelism for uniform distribution. For non-uniform distribution, it's a totally different case. Clearly, for non-uniform distribution, distributed memory parallelism of using moe MPI threads proves to be a bad way as it results in 4-5 times as many cycles. This is something that deals with the surrounding code, as the kernel of concern for these experiments gets launched as a single task, and has only OpenMP parallelism within it. Hence, it's out of scope of this study to explain this peculiarity with regard to non-uniform distribution having such baseline performance when run with more than 1 MPI tasks.

What is in the scope of this study is to analyze the impact of these run choices on the performance gain caused by the inserted prefetches. And that comparison reveals that more or less the relative performance improvement brought by inserted prefetches remains the same irrespective of the parallelization choice. This is expected too as in each case, there are 16 threads sharing the total work and prefetching works the same way, so should lead to similar performance gains.

5.6 Impact of Co-processor offloading

This is an interesting run configuration choice because of the reason that it deals with the indirect impact of the surrounding code on the kernel performance. No part of the VList kernel in question gets to offload on the coprocessor but parts of rest of the algorithm's steps do. Also, due to the way the overall code is structured, it is only possible to run it with 1 MPI task while coprocessor offloading in on, so for apples to apples comparison, it should be compared against the corresponding no processor offloading, 1 MPI task, 16 OMP thread run cases.

In terms of baseline performance, for uniform distribution, having coprocessor offloading improves the performance of kernel by about 2%. This can totally be attributed to indirect effects of having parts of surrounding code getting accelerated on MIC, resulting in possibly more cache and resources for the kernel. This difference is less pronounced with non-uniform distribution. This baseline shift causes a similar effect in terms of prefetch gains too, which are slightly more pronounced for uniform distribution without offloading than with offloading, and almost the same for non-uniform distribution.

In terms of the absolute number of cycles taken for the best prefetching combination for both uniform and non-uniform distribution, the numbers are almost the same for both co-processor offloading and no offloading. Therefore, it can be said that programmer inserted prefetches remain largely unaffected due to the indirect effects of coprocessor offloading on non-uniform distribution, and while they're affected nominally on uniform distribution, an improved baseline compensates for the change.

5.7 Impact of Compiler prefetching

With regard to the compiler prefetching, the most important thing to check is the baseline cycles, i.e. the no prefetch cycles. Out of 8 pairs of runs, there is only one pair which exhibits a significant difference in that baseline. That happens with Uniform distribution run with 16 MPI tasks without coprocessor offloading where with compiler prefetching on, the baseline performs 5% better. There is no apparent reason for this behavior as to why compiler prefetching becomes helpful for this particular configuration. It could possibly be an indirect effect of prefetching improving the performance of the surrounding code. That is also apparent from the fact that this run configuration pair is also the only one where the programmer inserted prefetches make a different impact in compiler prefetching off versus on case. Looking at Figure 14(b) and 15(b), while all the prefetching combinations improve the performance over baseline in varying degree complying with the general trend across runs, when compiler prefetching is off, when it's on, for some combinations, the performance is hurt. The reduction in performance can be attributed to higher than usual baseline. This shifted baseline also results in the preferred prefetching combinations registering less performance gains as compared to the compiler prefetching off case. There is no certain explanation for the shifted baseline but since all numbers are trimmed averages over 10 runs, there is no reason to doubt the veracity of the data either. It's an outlier.

However, except this one case, in general, compiler prefetching does not seem to make any difference to either the baseline numbers, or to the performance gain caused by programmer inserted prefetches against the baseline.

In the next section, the results for all these 16 run configurations are tabulated and plotted.

5.8 Result tables and graphs

Uniform Distribution, Compiler prefetching off, 1 MPI task, 16 OMP threads/task, No coprocessor offloading					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	910690228	890187105	904994736	813382025	805577876
To L2		872184914	894746093	827706953	838120182
To L3		871359035	863538677	843003230	863370050
To NTA		885392530	869261588	814912617	837429676

Table 6 - No. of cycles/thread for run configuration 1

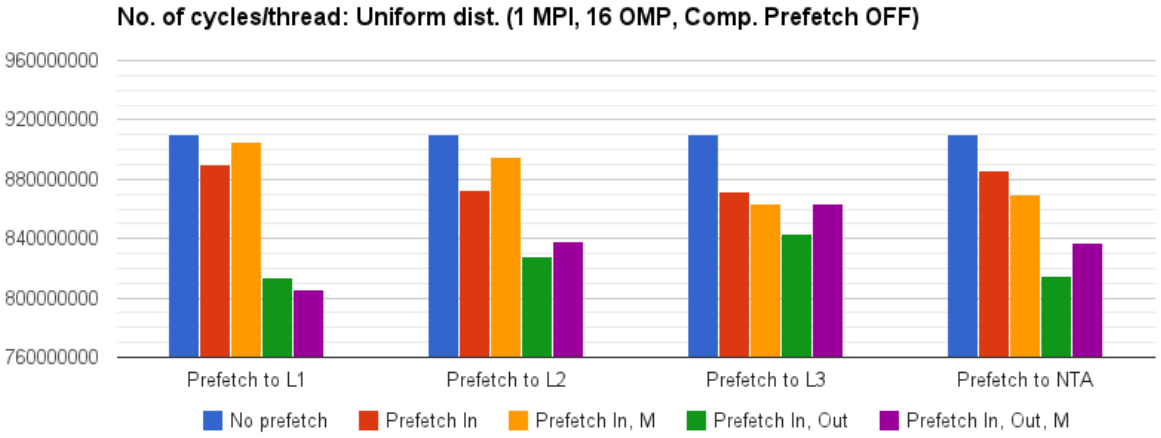

Figure 6(a) - No. of cycles/thread for run configuration 1

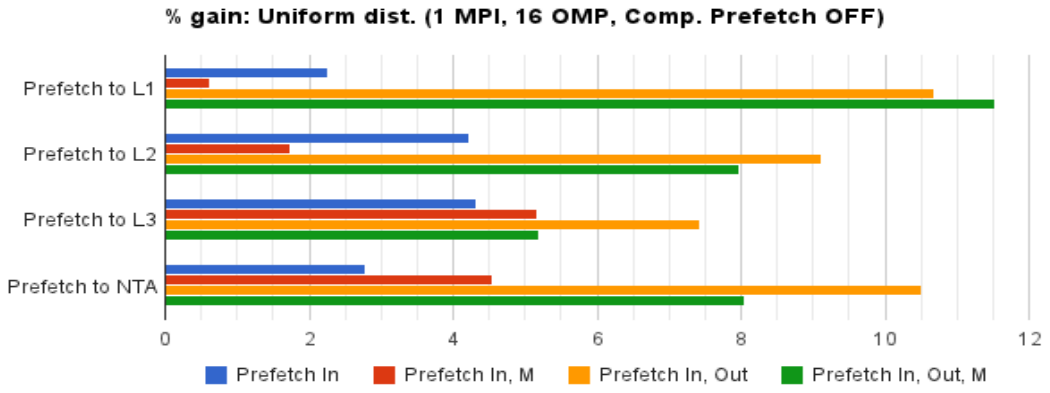

Figure 6(b) - Percentage gain (reduction in no. of cycles) for run configuration 1

Uniform Distribution, Compiler prefetching on 1 MPI task, 16 OpenMP threads/task, No coprocessor offloading					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	923121459	886954686	865685085	828099237	806611975
To L2		870189954	869126026	825903477	823958212
To L3		899146662	864530659	829791047	836722495
To NTA		870962281	877236662	832166812	808844077

Table 7 - No. of cycles/thread for run configuration 2

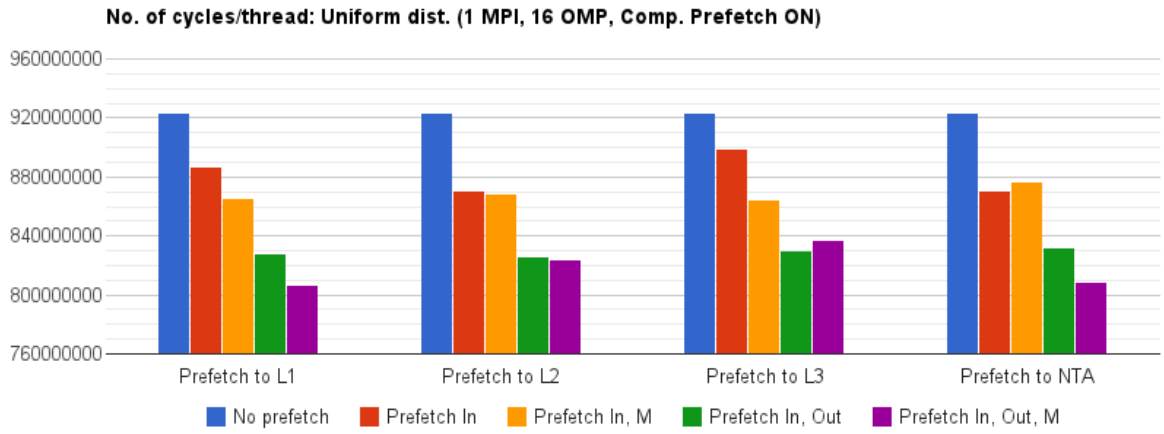

Figure 7(a) - No. of cycles/thread for run configuration 2

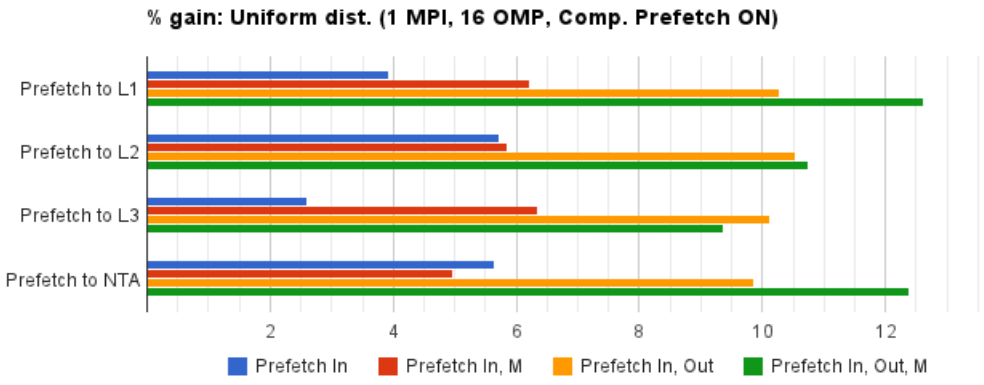

Figure 7(b) - Percentage gain (reduction in no. of cycles) for run configuration 2

Non-uniform Distribution, Compiler prefetching off 1 MPI task, 16 OpenMP threads/task, No coprocessor offloading					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	585525548	585410370	589778331	536397631	536415413
To L2		567900933	591813417	548788574	541069085
To L3		591012437	593237952	554709976	560555564
To NTA		581841270	589551645	529840825	533794800

Table 8 - No. of cycles/thread for run configuration 3

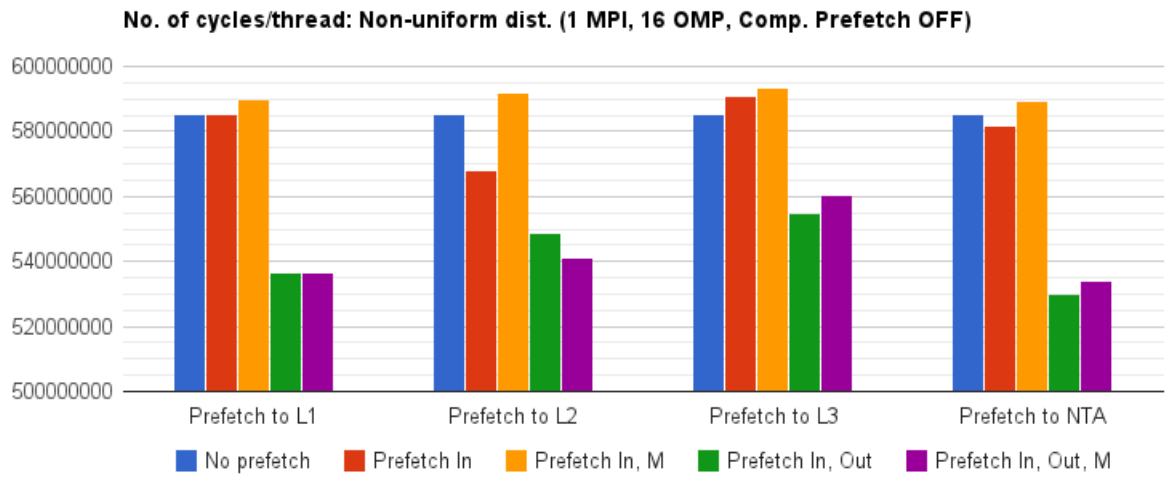

Figure 8(a) - No. of cycles/thread for run configuration 3

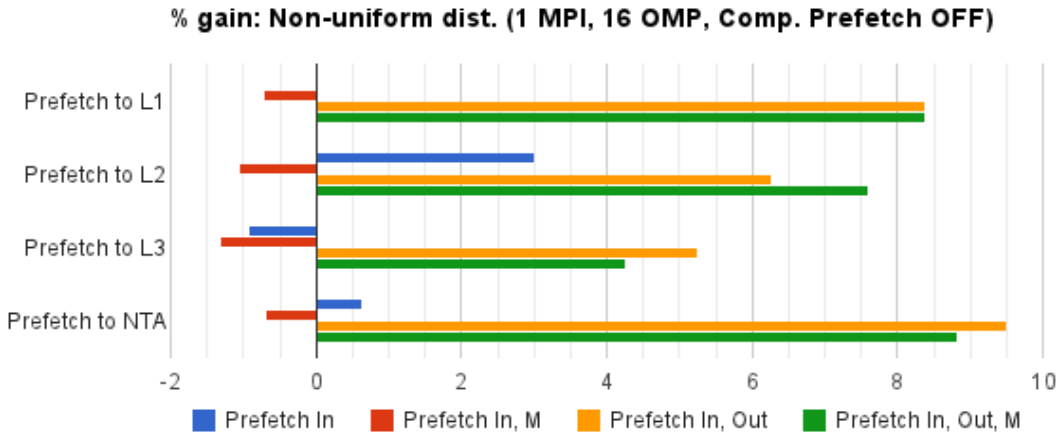

Figure 8(b) - Percentage gain (reduction in no. of cycles) for run configuration 3

Non-uniform Distribution, Compiler prefetching on 1 MPI task, 16 OpenMP threads/task, No coprocessor offloading					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	580158527	581782884	581419167	537565610	524479564
To L2		569617045	567717723	544640816	541850677
To L3		580541427	590075549	544634967	527423338
To NTA		581159757	577361946	544242122	522934693

Table 9 - No. of cycles/thread for run configuration 4

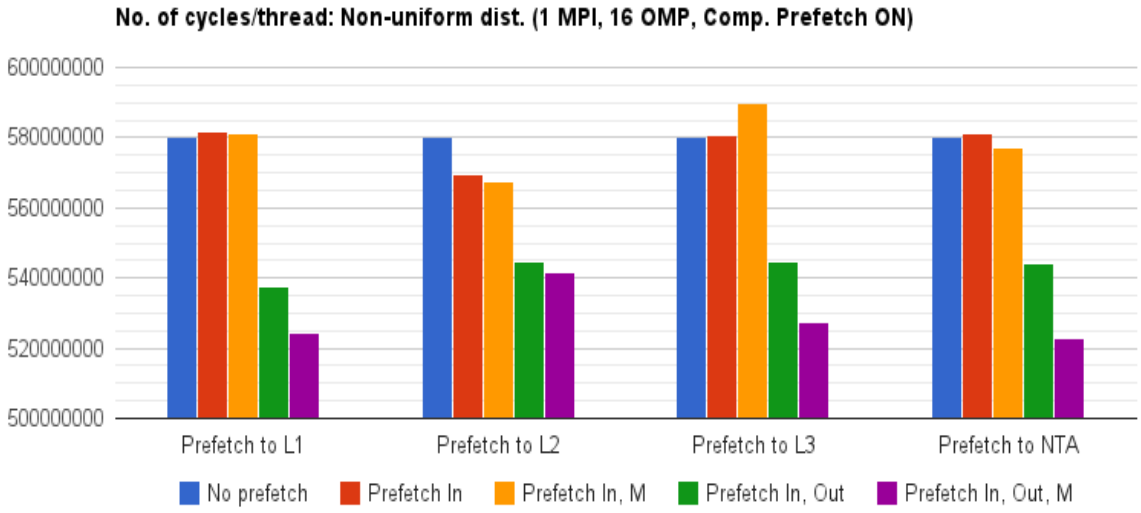

Figure 9(a) - No. of cycles/thread for run configuration 4

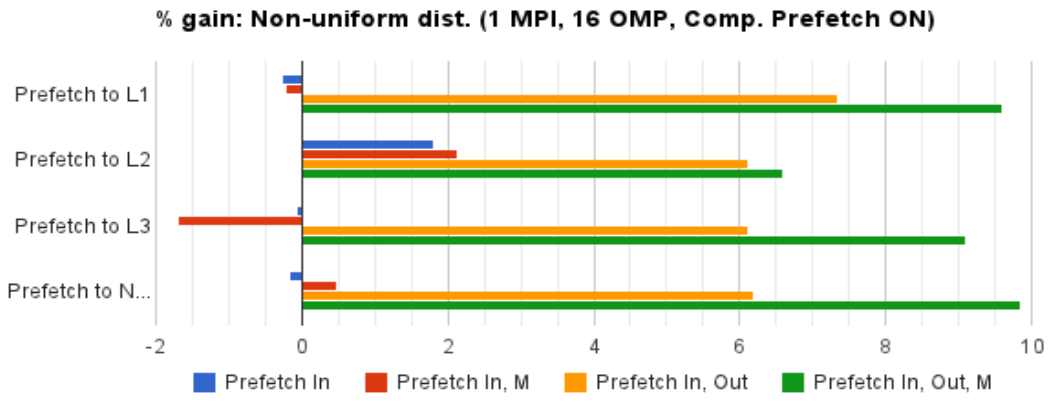

Figure 9(b) - Percentage gain (reduction in no. of cycles) for run configuration 4

Uniform Distribution, Compiler prefetching off 4 MPI tasks, 4 OpenMP threads/task, No coprocessor offloading					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	861552943	828071103	822131540	770931280	770829800
To L2		834141983	821182869	794805420	791047907
To L3		831186364	822954890	789903764	776795429
To NTA		824474602	822783422	779644684	762068794

Table 10 - No. of cycles/thread for run configuration 5

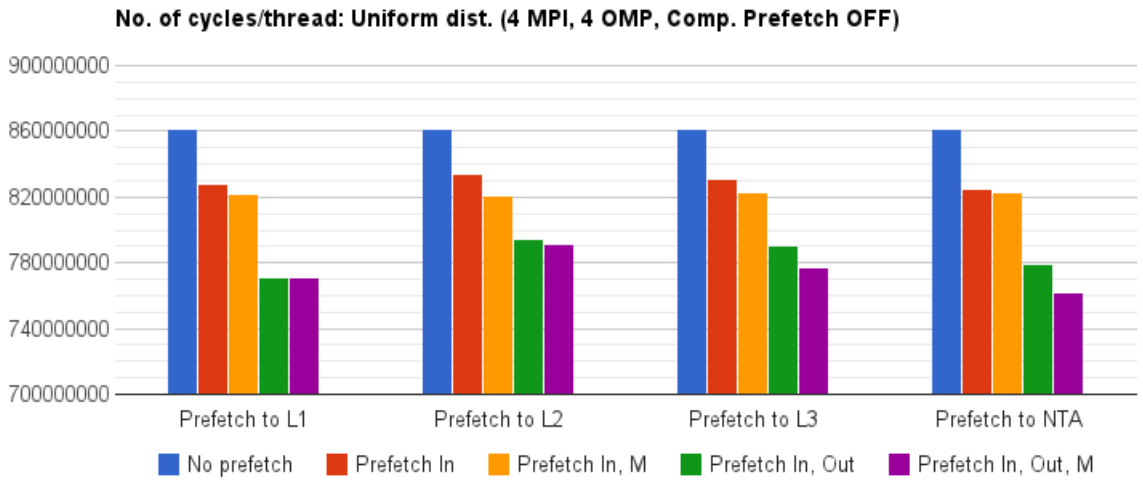

Figure 10(a) - No. of cycles/thread for run configuration 5

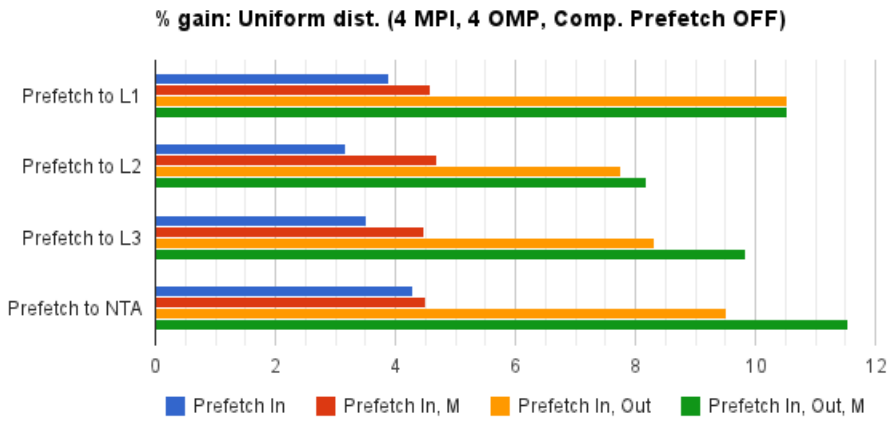

Figure 10(b) - Percentage gain (reduction in no. of cycles) for run configuration 5

Uniform Distribution, Compiler prefetching on 4 MPI tasks, 4 OpenMP threads/task, No coprocessor offloading					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	856304163	825329581	807386664	763337044	765418328
To L2		823979100	816563058	788341661	785329449
To L3		828714417	821044614	788136555	792211088
To NTA		828540243	812495344	779148500	764570545

Table 11 - No. of cycles/thread for run configuration 6

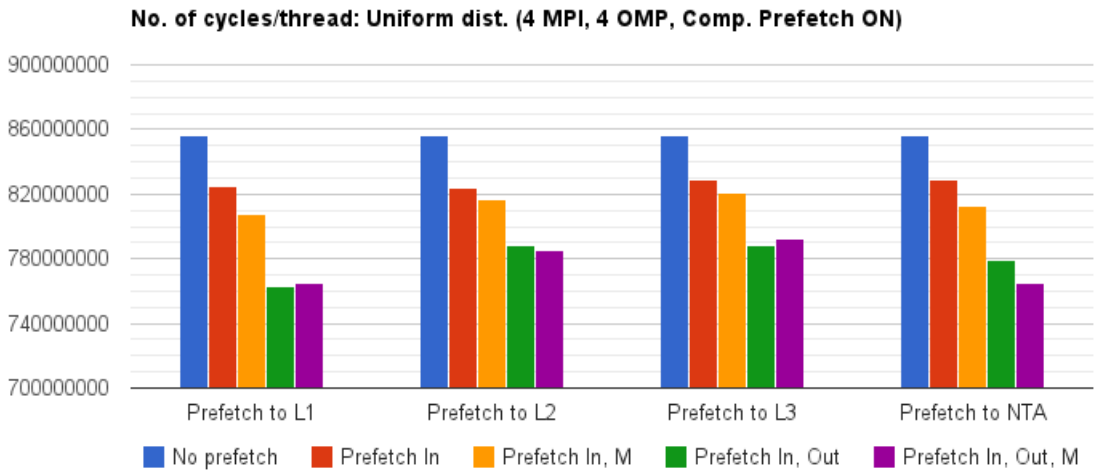

Figure 11(a) - No. of cycles/thread for run configuration 6

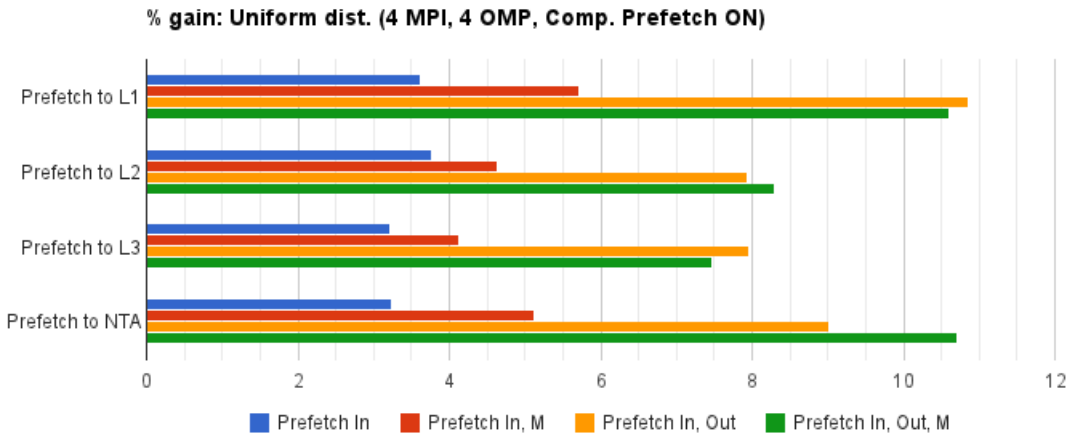

Figure 11(b) - Percentage gain (reduction in no. of cycles) for run configuration 6

Non-uniform Distribution, Compiler prefetching off 4 MPI tasks, 4 OpenMP threads/task, No coprocessor offloading					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	2692200736	2607237244	2586955006	2476054782	2452162642
To L2		2604391502	2598818781	2522211248	2509300542
To L3		2619871098	2589067698	2519774272	2500372906
To NTA		2605647314	2600025509	2459153446	2450824145

Table 12 - No. of cycles/thread for run configuration 7

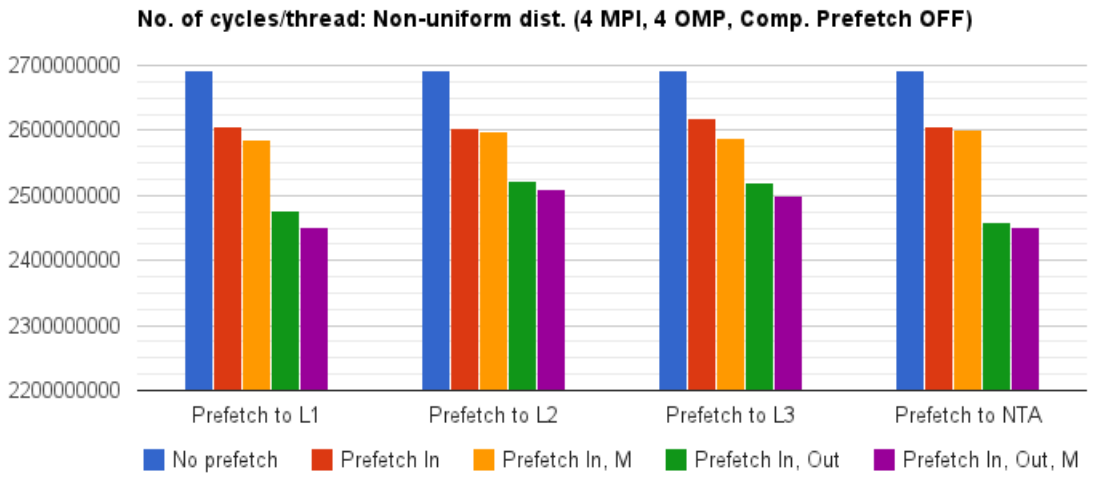

Figure 12(a) - No. of cycles/thread for run configuration 7

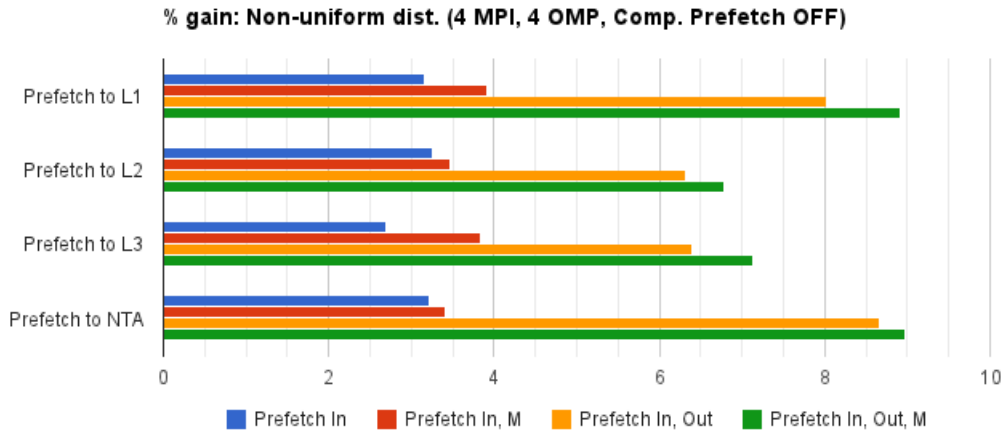

Figure 12(b) - Percentage gain (reduction in no. of cycles) for run configuration 7

Non-uniform Distribution, Compiler prefetching on 4 MPI tasks, 4 OpenMP threads/task, No coprocessor offloading					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	2686209580	2600029065	2585300081	2458011688	2443226047
To L2		2604692366	2588299467	2519532033	2503456685
To L3		2604080247	2613487092	2529253910	2517189108
To NTA		2612387307	2577576503	2457885961	2460558988

Table 13 - No. of cycles/thread for run configuration 8

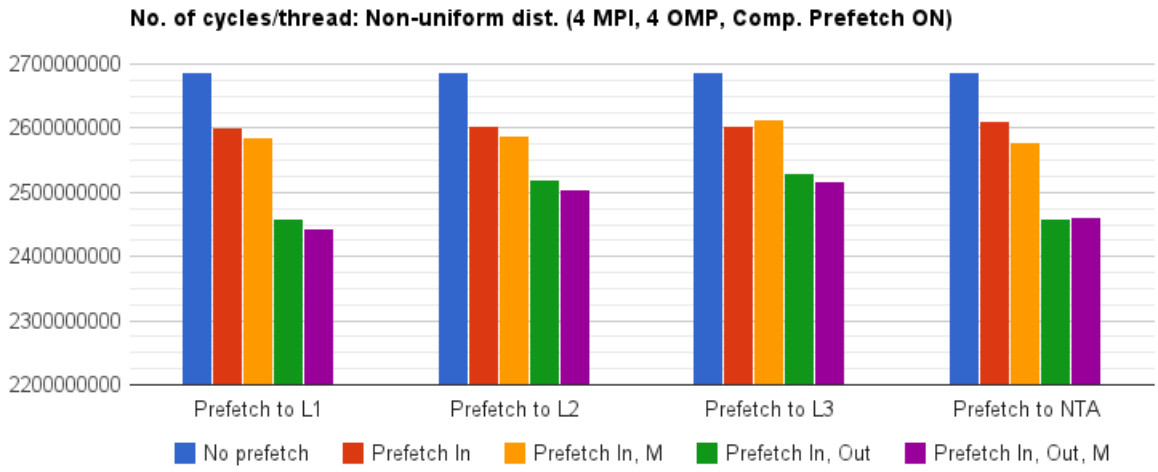

Figure 13(a) - No. of cycles/thread for run configuration 8

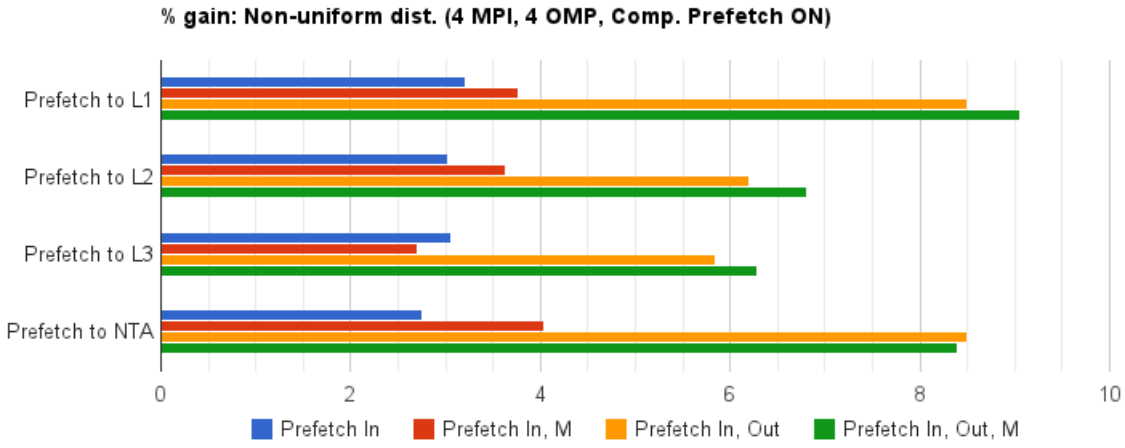

Figure 13(b) - Percentage gain (reduction in no. of cycles) for run configuration 8

Uniform Distribution, Compiler prefetching off 16 MPI tasks, 1 OpenMP threads/task, No coprocessor offloading					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	818590707	780420295	788755148	676736573	726061328
To L2		784199460	778057876	746167243	757623671
To L3		804639318	792194888	762975476	715500402
To NTA		785157955	796414733	735356392	727481281

Table 14 - No. of cycles/thread for run configuration 9

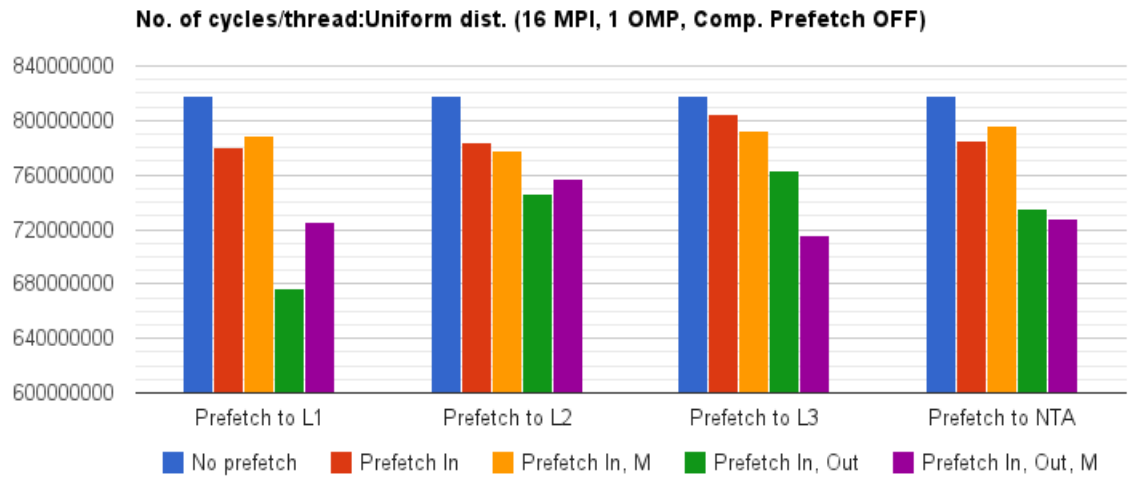

Figure 14(a) - No. of cycles/thread for run configuration 9

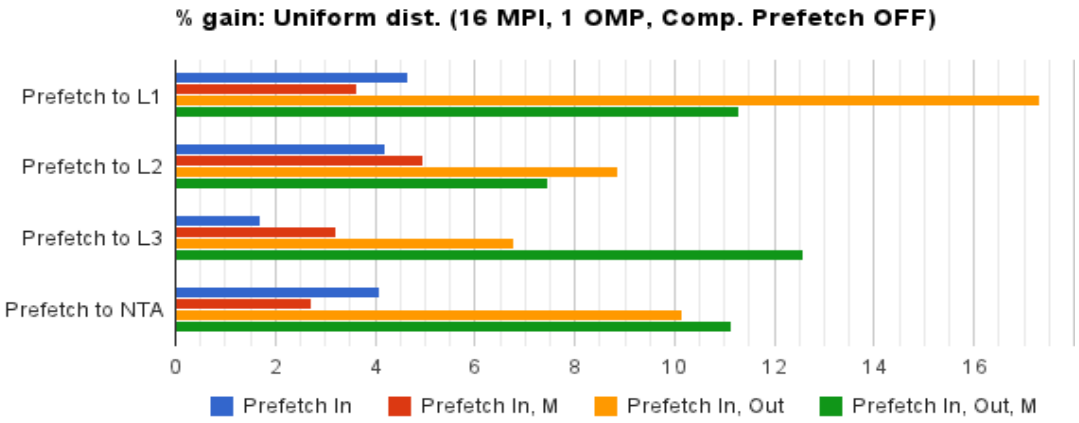

Figure 14(b) - Percentage gain (reduction in no. of cycles) for run configuration 9

Uniform Distribution, Compiler prefetching on 16 MPI tasks, 1 OpenMP threads/task, No coprocessor offloading					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	778079949	799506258	777568677	720362061	724550227
To L2		803117329	782626923	726324415	705618299
To L3		783115452	742796331	746999163	753926871
To NTA		750622889	793244770	729227175	720189404

Table 15 - No. of cycles/thread for run configuration 10

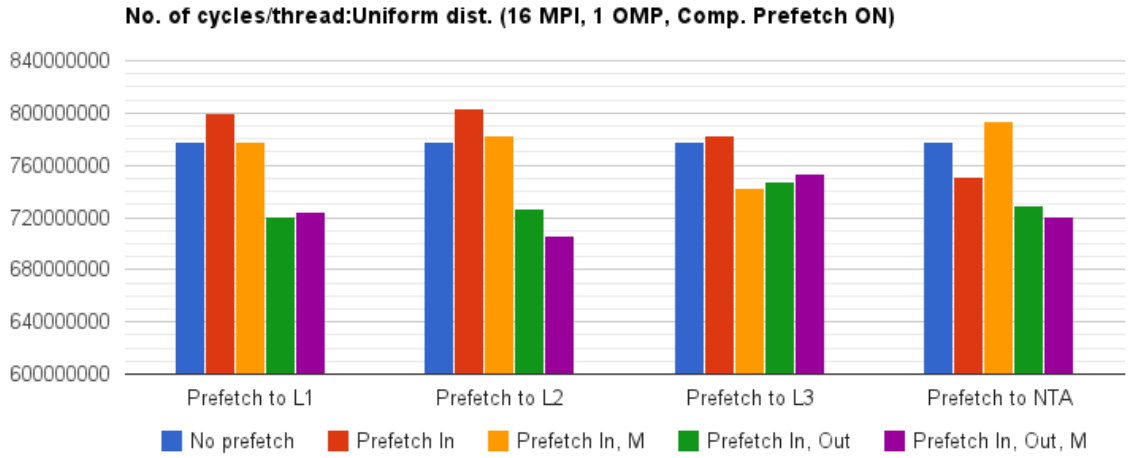

Figure 15(a) - No. of cycles/thread for run configuration 10

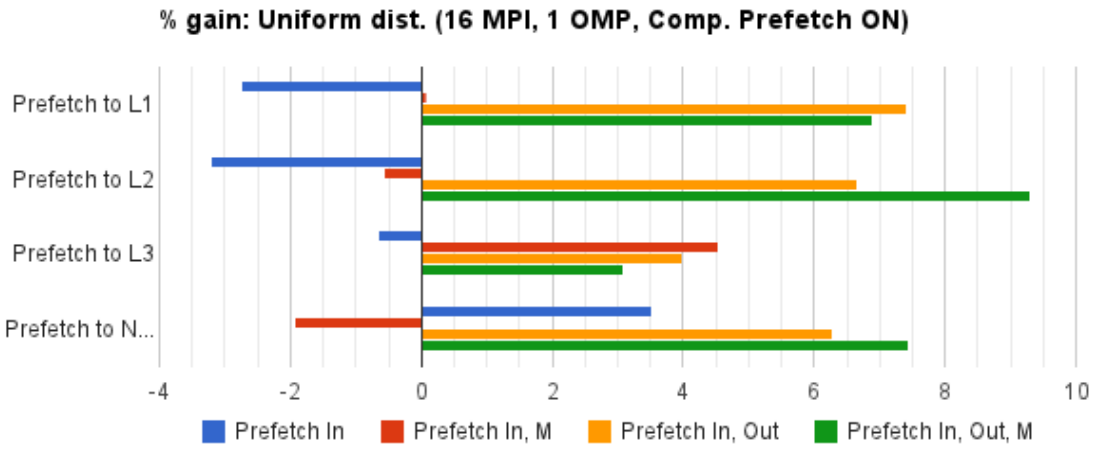

Figure 15(b) - Percentage gain (reduction in no. of cycles) for run configuration 10

Non-uniform Distribution, Compiler prefetching off 16 MPI tasks, 1 OpenMP thread/task, No coprocessor offloading					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	2574794528	2447734995	2453215485	2306941143	2267227094
To L2		2482538144	2454004520	2375136413	2365039945
To L3		2494890141	2483171305	2376823089	2405799508
To NTA		2507980368	2440409433	2282875881	2286068814

Table 16 - No. of cycles/thread for run configuration 11

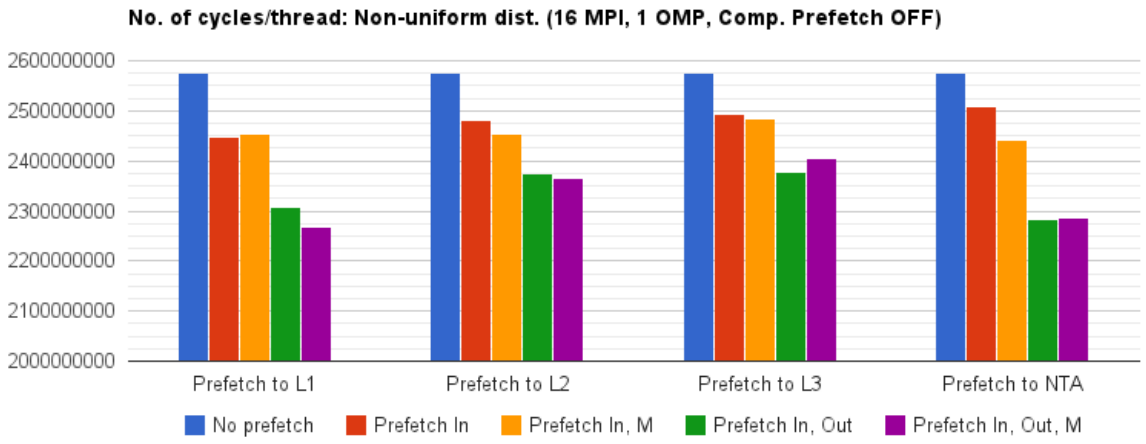

Figure 16(a) - No. of cycles/thread for run configuration 11

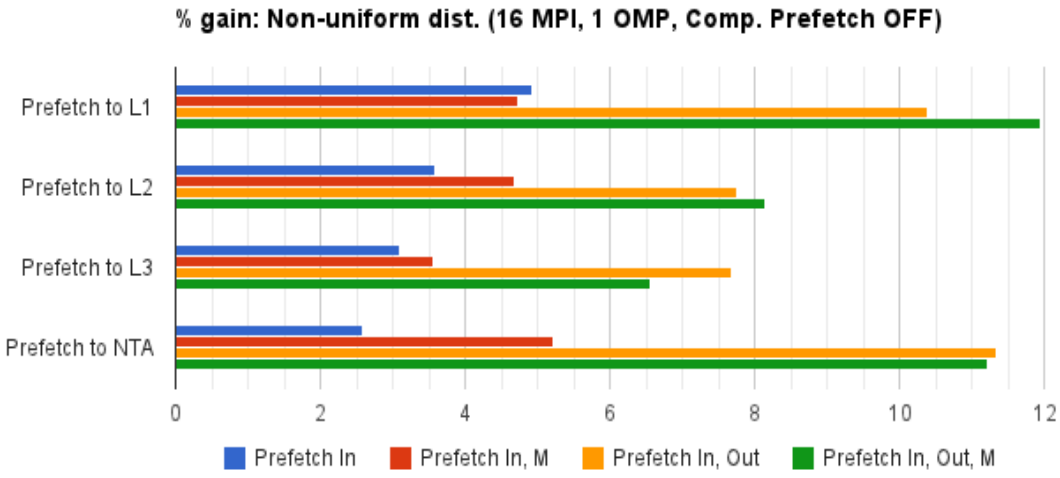

Figure 16(b) - Percentage gain (reduction in no. of cycles) for run configuration 11

Non-uniform Distribution, Compiler prefetching on 16 MPI tasks, 1 OpenMP threads/task, No coprocessor offloading					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	2600228519	2469273119	2429393288	2313591094	2296131604
To L2		2499694563	2477001727	2377432739	2388893188
To L3		2503980170	2468614226	2364781199	2376956848
To NTA		2474802184	2442371445	2283756840	2314909771

Table 17 - No. of cycles/thread for run configuration 12

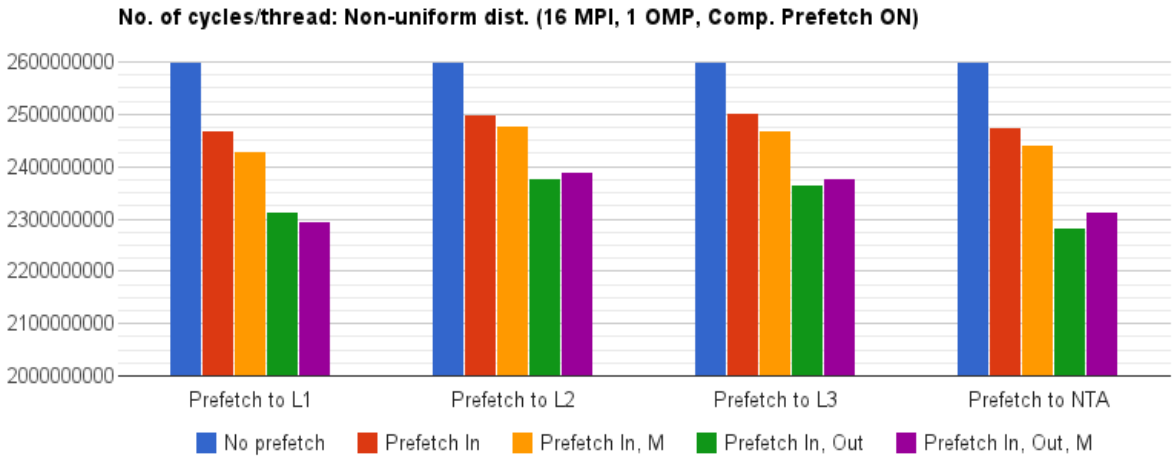

Figure 17(a) - No. of cycles/thread for run configuration 12

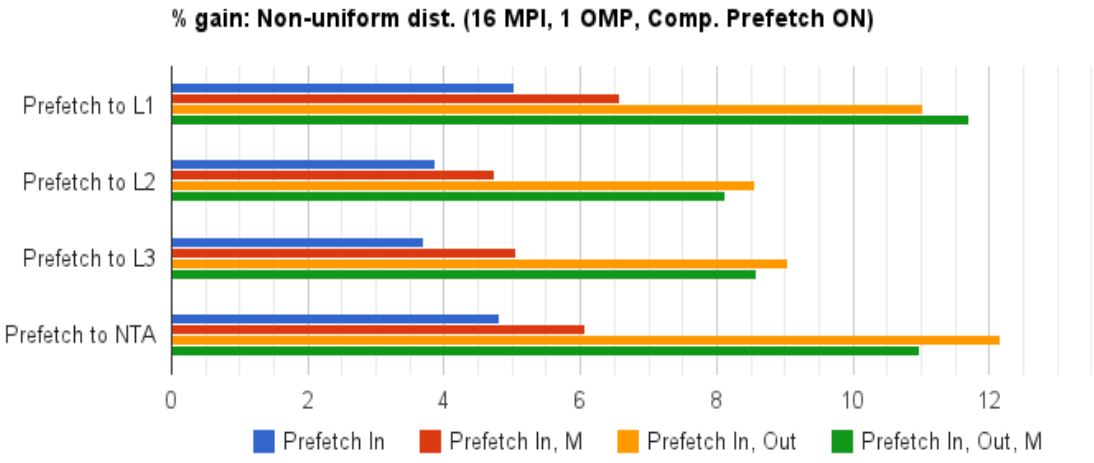

Figure 17(b) - Percentage gain (reduction in no. of cycles) for run configuration 12

Uniform Distribution, Compiler prefetching off 1 MPI task, 16 OpenMP threads/task, Coprocessor offloading on					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	894213325	870450169	865500577	813513471	808726915
To L2		872397249	866834890	825470097	823003772
To L3		870717836	865200227	827762715	822946564
To NTA		870830833	867296145	817165217	809265066

Table 18 - No. of cycles/thread for run configuration 13

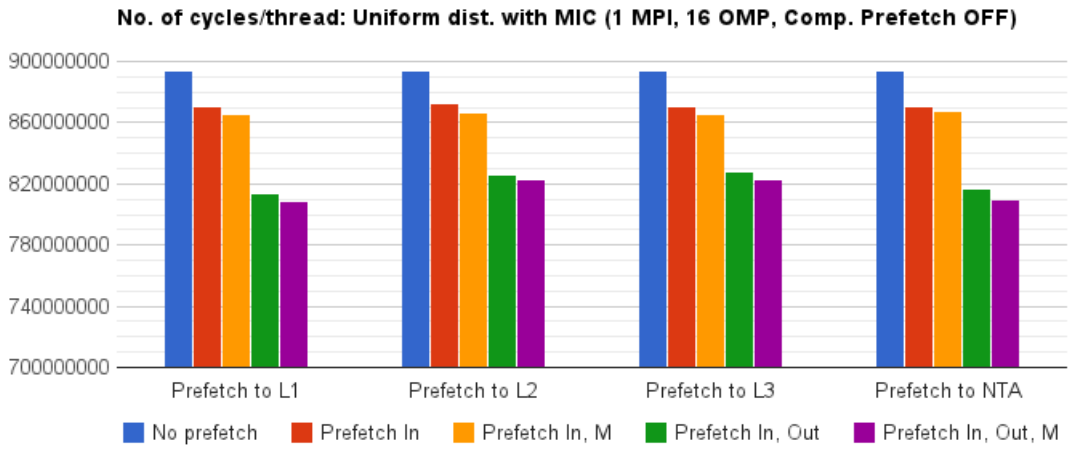

Figure 18(a) - No. of cycles/thread for run configuration 13

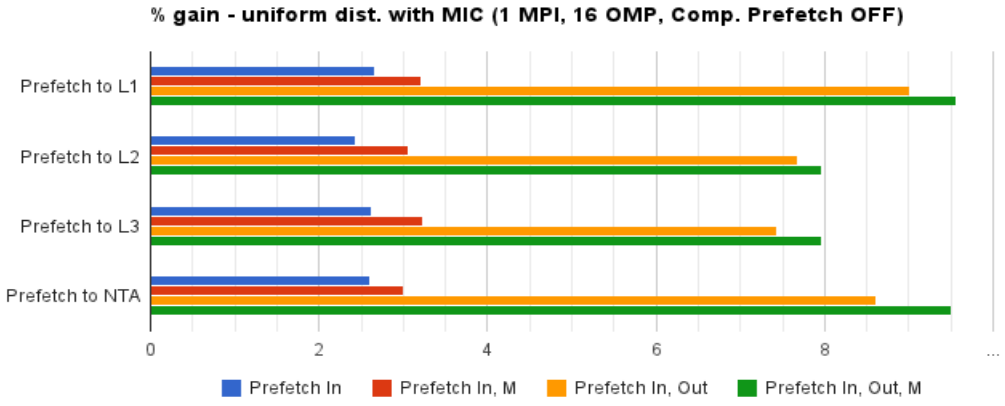

Figure 18(b) - Percentage gain (reduction in no. of cycles) for run configuration 13

Uniform Distribution, Compiler prefetching on, 1 MPI task, 16 OpenMP threads/task, Coprocessor offloading on					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	899033490	872957902	865795490	813663817	806967848
To L2		871871483	865754737	830842283	821663403
To L3		870607904	867115921	828467887	822120071
To NTA		871732007	870604436	814345490	808450165

Table 19 - No. of cycles/thread for run configuration 14

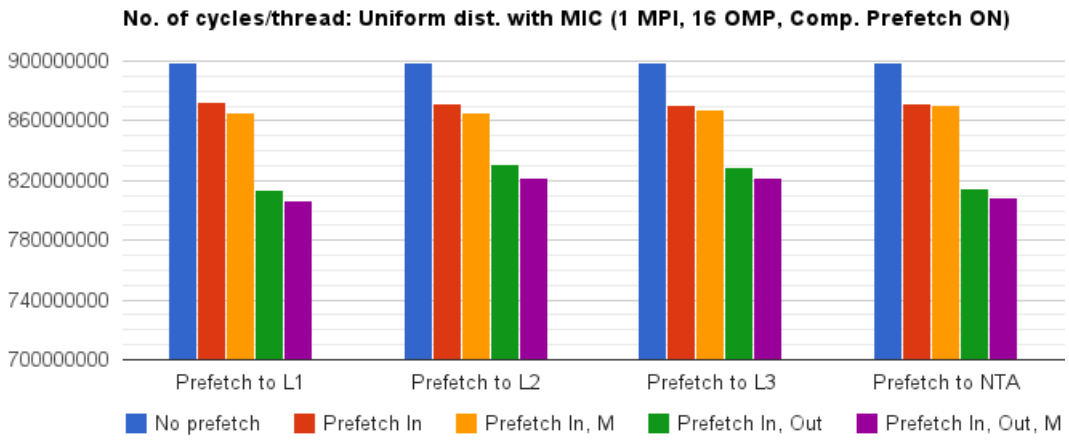

Figure 19(a) - No. of cycles/thread for run configuration 14

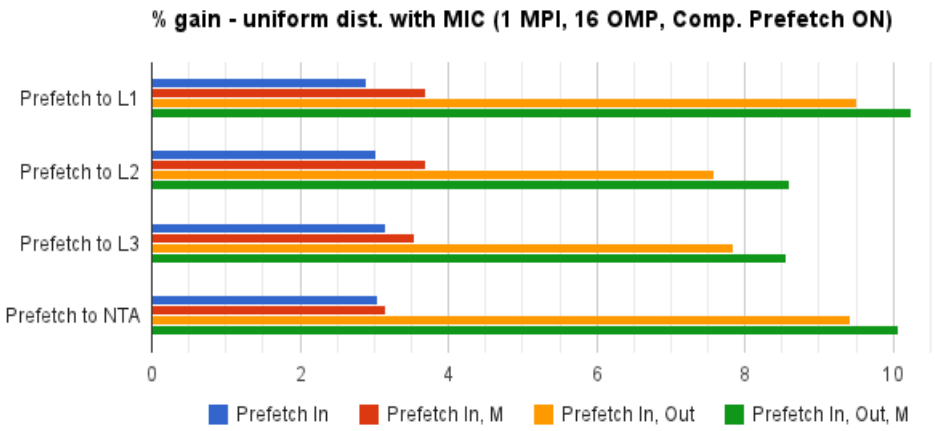

Figure 19(b) - Percentage gain (reduction in no. of cycles) for run configuration 14

Non-uniform Distribution, Compiler prefetching off 1 MPI task, 16 OpenMP threads/task, Coprocessor offloading on					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	581649228	569310928	569989899	526162707	525968804
To L2		567457437	580449652	529019240	530701060
To L3		577516425	567070481	539134956	531184698
To NTA		580574981	567215305	531632753	528823809

Table 20 - No. of cycles/thread for run configuration 15

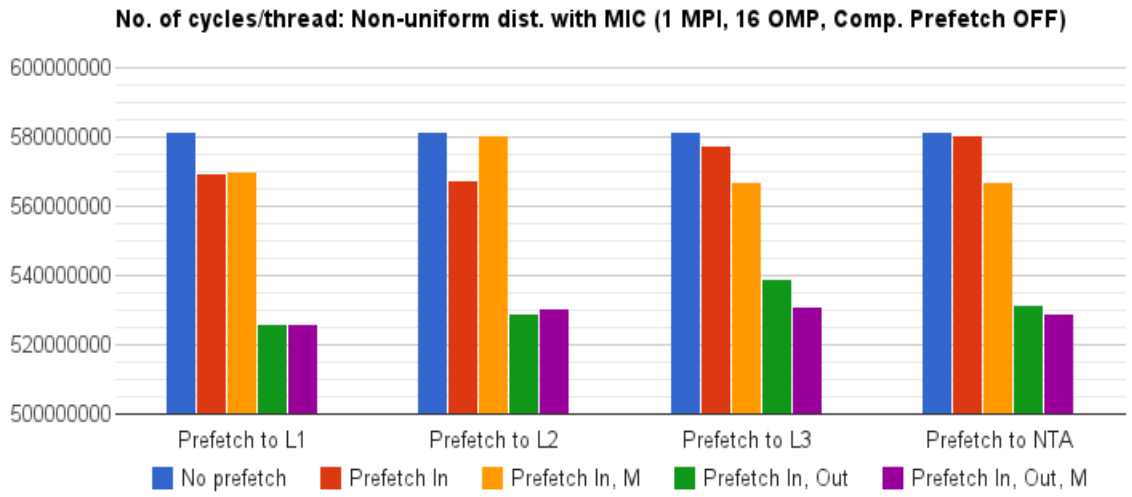

Figure 20(a) - No. of cycles/thread for run configuration 15

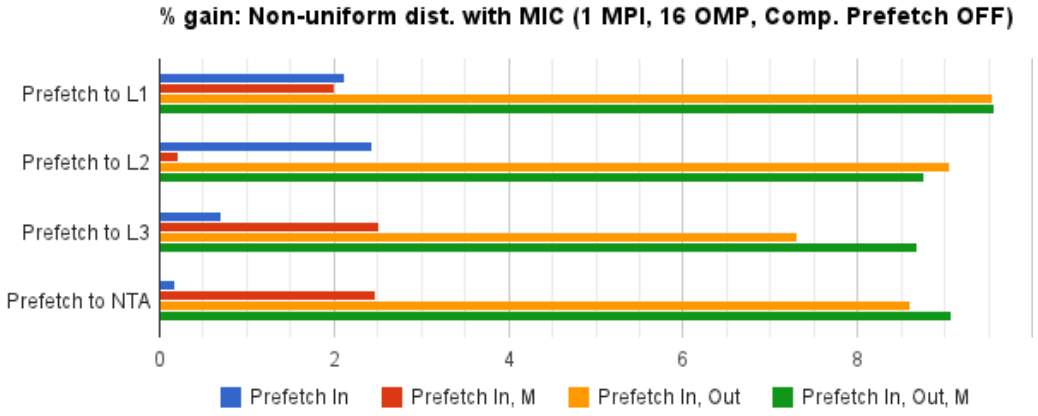

Figure 20(b) - Percentage gain (reduction in no. of cycles) for run configuration 15

Non-uniform Distribution, Compiler prefetching on 1 MPI task, 16 OpenMP threads/task, Coprocessor offloading on					
Prefetch	**None**	**IN array only**	**IN, M arrays**	**IN, OUT arrays**	**IN, OUT, M arrays**
To L1	579562037	566738786	566669405	529775230	522980038
To L2		568893065	569328925	541889106	535481411
To L3		581155112	571147202	533073555	527217520
To NTA		570478481	568829708	533495562	528143228

Table 21 - No. of cycles/thread for run configuration 16

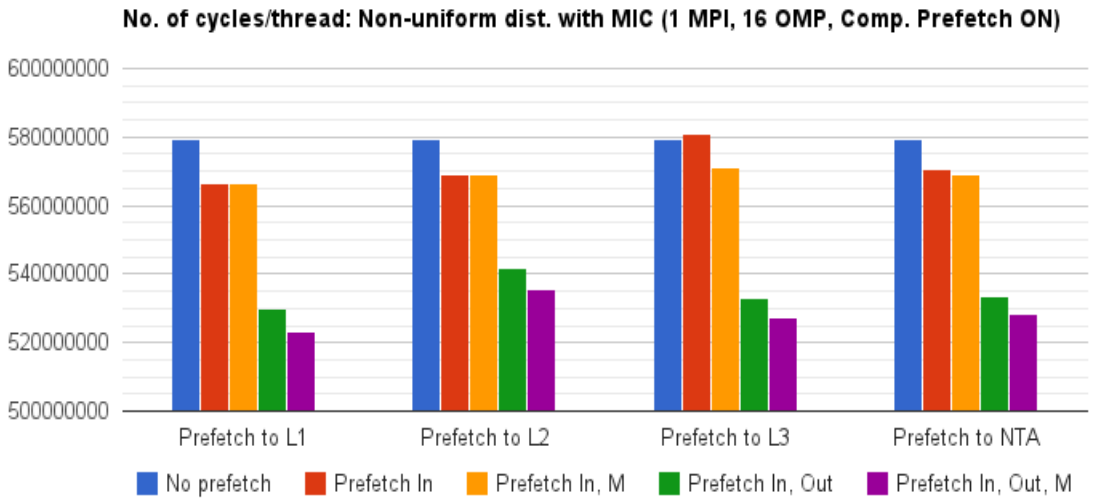

Figure 21(a) - No. of cycles/thread for run configuration 16

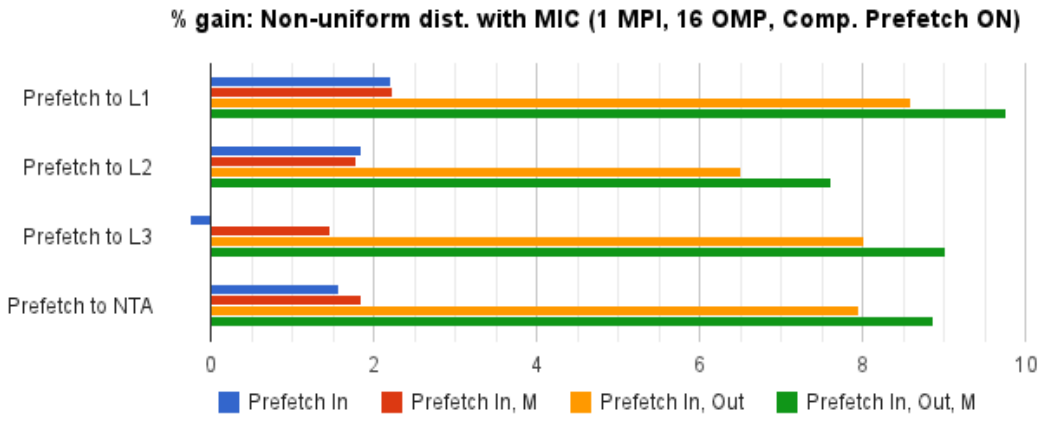

Figure 21(b) - Percentage gain (reduction in no. of cycles) for run configuration 16

6 CONCLUSION

In this study, this author presented a case for programmer inserted prefetches. An introduction of hardware and prefetching techniques was given and a case for programmer inserted prefetches was identified for programs where access patterns are such which cannot be identified by hardware prefetchers and/or they're indirect so they cannot be identified by compiler either. With the help of an FMM VList case study, it was demonstrated that for cases like this, programmer inserted prefetches are a great solution. Several prefetching choices were explored and empirically compared across 16 different run configurations. The author was able to attain consistent performance gain of 9-12% across most run configurations, and exceeding 12% for some run configurations for their respective best prefetching combinations. Prefetching all 3 data structures and prefetching those to L1 cache, was identified as the one prefetching combination that performed best for majority of run configurations, and second best for majority of the rest. This combination resulted in a performance gain of 10.14% averaged across all run configurations. The broader trends with regard to prefetching gains were listed and analyzed along with variations caused by different run configuration parameters.

In conclusion, it can be said that scientific applications such as the one taken as a case study for this research, do have irregular access patterns where both hardware prefetchers and compilers fail to help, and it is in such applications, that programmer inserted prefetches can make a significant positive difference and bring about precious performance gains.

Appendix A PAPI EventSet Creation and Calls

```
#include "/opt/apps/papi/5.2RC/include/papi.h"
#include "/opt/apps/papi/5.2RC/include/papiStdEventDefs.h"
int EventSet = PAPI_NULL;
long_long values[3];
int main () {
   PAPI_event_info_t evinfo;
   int eventlist[] =
   {
#if X // NOT MORE THAN 5 EVENTS CAN BE MEASURED AT ONCE
     PAPI_L1_DCM,      PAPI_L2_DCM,      PAPI_L3_TCM,
#else
     PAPI_LD_INS,      PAPI_TOT_CYC,     PAPI_TOT_INS,
#endif
     0
   };

  int retval = PAPI_library_init(PAPI_VER_CURRENT);
  if (PAPI_thread_init(omp_get_thread_num) != PAPI_OK) std::cout <<
"handle_error0" << std::endl;
  if (retval != PAPI_VER_CURRENT && retval > 0)  std::cout << "PAPI library
version mismatch!\n";
  if (retval < 0) std::cout << "Initialization error!\n";
  if (PAPI_create_eventset(&EventSet) != PAPI_OK)  std::cout <<
"handle_error1" << std::endl;
  for (int i=0; eventlist[i] != 0; i++) {
    if (PAPI_add_event(EventSet, eventlist[i]) != PAPI_OK) std::cout <<
"handle_error2." << i << std::endl;
  }

/* PAPI EVENTSET CREATION CODE ABOVE */
/* MAIN FUNCTION CODE HERE */
/* PAPI EVENTSET PRINT CODE BELOW */

  std::ofstream out("out.txt");
  std::streambuf *coutbuf = std::cout.rdbuf();
  std::cout.rdbuf(out.rdbuf()); //redirect std::cout to out.txt!
  for (int i=0; eventlist[i] != 0; i++) {
    PAPI_get_event_info(eventlist[i], &evinfo);
    std::cout << evinfo.symbol << "=\t" << values[i] << "\n";
  }
  std::cout.rdbuf(coutbuf);
} // end of main
```

```
if (PAPI_start(EventSet) != PAPI_OK) std::cout << "handle_error3" <<
std::endl;
/* CALL TO KERNEL TO BE PROFILED */
if (PAPI_stop(EventSet, values) != PAPI_OK) std::cout << "handle_error4" <<
std::endl;
```

Appendix B Programmer Inserted Prefetches

```
    for(size_t    blk1=0;    blk1<blk1_cnt;    blk1++)
    for(size_t       k=a;       k<      b;      k++)
    for(size_t mat_indx=0; mat_indx< mat_cnt;mat_indx++){
      size_t interac_blk1 = blk1*mat_cnt+mat_indx;
      size_t interac_dsp0 =
(interac_blk1==0?0:interac_dsp[interac_blk1-1]);
      size_t interac_dsp1 =                interac_dsp[interac_blk1  ] ;
      size_t interac_cnt  = interac_dsp1-interac_dsp0;
      Real_t** IN = IN_ + 2*V_BLK_SIZE*interac_blk1;
      Real_t** OUT= OUT_+ 2*V_BLK_SIZE*interac_blk1;
      Real_t* M = precomp_mat[mat_indx] + k*chld_cnt*chld_cnt*2;
#if PREF_M
      if (mat_indx +1 < mat_cnt) {
        _mm_prefetch(((char *)(precomp_mat[mat_indx+1] +
k*chld_cnt*chld_cnt*2)), _MM_HINT_NTA);
        _mm_prefetch(((char *)(precomp_mat[mat_indx+1] +
k*chld_cnt*chld_cnt*2) + 64), _MM_HINT_NTA);
      }
#endif
      for(int in_dim=0;in_dim<ker_dim0;in_dim++)
      for(int ot_dim=0;ot_dim<ker_dim1;ot_dim++){
        for(size_t j=0;j<interac_cnt;j+=2){
          Real_t* M_   = M;
          Real_t* IN0  = IN [j+0] + (in_dim*M_dim+k)*chld_cnt*2;
          Real_t* IN1  = IN [j+1] + (in_dim*M_dim+k)*chld_cnt*2;
          Real_t* OUT0 = OUT[j+0] + (ot_dim*M_dim+k)*chld_cnt*2;
          Real_t* OUT1 = OUT[j+1] + (ot_dim*M_dim+k)*chld_cnt*2;
#if PREF_IN
          if (j+2 < interac_cnt) {
            _mm_prefetch(((char *)(IN[j+2] + (in_dim*M_dim+k)*chld_cnt*2)),
_MM_HINT_NTA);
            _mm_prefetch(((char *)(IN[j+2] + (in_dim*M_dim+k)*chld_cnt*2) +
64), _MM_HINT_NTA);
            _mm_prefetch(((char *)(IN[j+3] + (in_dim*M_dim+k)*chld_cnt*2)),
_MM_HINT_NTA);
            _mm_prefetch(((char *)(IN[j+3] + (in_dim*M_dim+k)*chld_cnt*2) +
64), _MM_HINT_NTA);
          }
#endif
#if PREF_OUT
          if (j+2 < interac_cnt) {
            _mm_prefetch(((char *)(OUT[j+2] +
(ot_dim*M_dim+k)*chld_cnt*2)), _MM_HINT_NTA);
            _mm_prefetch(((char *)(OUT[j+2] + (ot_dim*M_dim+k)*chld_cnt*2) +
64), _MM_HINT_NTA);
            _mm_prefetch(((char *)(OUT[j+3] +
(ot_dim*M_dim+k)*chld_cnt*2)), _MM_HINT_NTA);
            _mm_prefetch(((char *)(OUT[j+3] + (ot_dim*M_dim+k)*chld_cnt*2) +
64), _MM_HINT_NTA);
          }
#endif
```

Appendix C.1

Dataset for Uniform Distribution with Compiler prefetching off, run with 1 MPI task, 16 OpenMP threads/task with no coprocessor offloading

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	39382047	11437715	6558779	257580012	890187105	1138851100	2508531367	2.25%
L1_IOM101	40219367	11721880	6522710	267467736	904994736	1170275155	2510431460	0.63%
L1_IOM110	41414027	11609441	6428982	266173560	813382025	1134407195	2503626513	10.69%
L1_IOM111	41482088	11464788	6350162	267599810	805577876	1136974888	2504262669	11.54%
L2_IOM100	38275314	11366451	6099773	248369202	872184914	1116333230	2506846016	4.23%
L2_IOM101	38706669	11638370	5986141	261414000	894746093	1155475165	2506761156	1.75%
L2_IOM110	38427992	11523267	5915361	266196959	827706953	1134462756	2504159044	9.11%
L2_IOM111	38382712	11521751	5730941	273346205	838120182	1151021690	2504393921	7.97%
L3_IOM100	38187334	11361401	6117797	248871756	871359035	1117561517	2506323551	4.32%
L3_IOM101	38646289	11583941	5985405	249889085	863538677	1127300449	2506830436	5.18%
L3_IOM110	38387912	11487766	5913674	272812426	843003230	1150635351	2504046640	7.43%
L3_IOM111	38387299	11429113	5717579	284057859	863370050	1177208744	2504360813	5.20%
NTA_IOM100	39774206	11888200	6625212	254453987	885392530	1131207764	2508196652	2.78%
NTA_IOM101	40369874	12065690	6578188	251558686	869261588	1131382056	2509086301	4.55%
NTA_IOM110	41037275	15238248	6655987	265956929	814912617	1133876120	2503848484	10.52%
NTA_IOM111	40896919	15250271	6642499	275053984	837429676	1155194813	2504494973	8.04%
No Prefetch	38490819	11528379	6670958	242159177	910690228	1121749584	2505497753	0.00%

Appendix C.2

Dataset for Uniform Distribution with Compiler prefetching on, run with 1 MPI task, 16 OpenMP threads/task with no coprocessor offloading

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	39592667	11450847	6562843	251245483	886954686	1123361726	2509517431	3.92%
L1_IOM101	40122583	11675547	6546864	250053663	865685085	1127703001	2509720963	6.22%
L1_IOM110	41645833	11519698	6426834	272568242	828099237	1150040116	2503552328	10.29%
L1_IOM111	41574496	11609308	6375550	267091724	806611975	1135732735	2504201993	12.62%
L2_IOM100	38257414	11399007	6128831	247864345	870189954	1115099248	2506164089	5.73%
L2_IOM101	38821122	11638651	5971871	250557437	869126026	1128933295	2507158269	5.85%
L2_IOM110	38625679	11556748	5915276	266414688	825903477	1134994945	2503845613	10.53%
L2_IOM111	38444391	11467983	5728579	267629316	823958212	1137044462	2504232225	10.74%
L3_IOM100	38415806	11383630	6099012	259866825	899146662	1144441088	2507029464	2.60%
L3_IOM101	38760227	11578680	5973563	249855274	864530659	1127218031	2506879425	6.35%
L3_IOM110	38563928	11500320	5907023	266116758	829791047	1134266984	2504029605	10.11%
L3_IOM111	38424645	11457504	5724795	272727795	836722495	1149509415	2504393199	9.36%
NTA_IOM100	39813209	11886604	6630120	248509918	870962281	1116677865	2508864587	5.65%
NTA_IOM101	40536825	12123140	6582487	256037947	877236662	1142333348	2510005636	4.97%
NTA_IOM110	41078186	15268807	6660511	273204497	832166812	1151595515	2503825472	9.85%
NTA_IOM111	41137099	15304813	6640645	267278203	808844077	1136187522	2504458763	12.38%
No Prefetch	38488646	11625507	6689717	241523244	923121459	1120195427	2505796045	0.00%

Appendix C.3

Dataset for Non-uniform Distribution with Compiler prefetching off, run with 1 MPI task, 16 OpenMP threads/task with no coprocessor offloading

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	25695761	6612366	4100117	175004633	585410370	767842617	1675143688	0.02%
L1_IOM101	26114454	6682556	4099480	171076551	589778331	763084620	1676022699	-0.73%
L1_IOM110	26870326	6666680	4011723	183418493	536397631	771307620	1671784119	8.39%
L1_IOM111	27007589	6614928	3989914	183102004	536415413	769922857	1672220071	8.39%
L2_IOM100	24765734	6453888	3778545	166369492	567900933	746732416	1673638581	3.01%
L2_IOM101	25155128	6618529	3736379	178994615	591813417	782443389	1674152439	-1.07%
L2_IOM110	25020788	6498100	3533919	181834560	548788574	767434506	1672007168	6.27%
L2_IOM111	24913921	6534554	3450125	182478978	541069085	768398704	1672219216	7.59%
L3_IOM100	24709892	6449675	3807649	174740157	591012437	767196136	1673753572	-0.94%
L3_IOM101	25107467	6646783	3742869	182195760	593237952	790269149	1674062884	-1.32%
L3_IOM110	24969223	6561181	3523360	186624468	554709976	779145222	1672001415	5.26%
L3_IOM111	24897180	6532418	3440464	188218721	560555564	782429589	1672192504	4.26%
NTA_IOM100	25811514	6779401	4152355	174334773	581841270	766204996	1674924011	0.63%
NTA_IOM101	26386018	6941601	4124487	175718086	589551645	774432135	1675575271	-0.69%
NTA_IOM110	26595637	9025100	4189122	178058174	529840825	758204057	1671825039	9.51%
NTA_IOM111	26751488	9057964	4180787	178392654	533794800	758408059	1672326765	8.83%
No Prefetch	25056523	6621272	4172231	156755492	585525548	737072658	1673456467	0.00%

Appendix C.4

Dataset for Non-uniform Distribution with Compiler prefetching on, run with 1 MPI task, 16 OpenMP threads/task with no coprocessor offloading

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	25621656	6633261	4098441	173523493	581782884	764221714	1675147564	-0.28%
L1_IOM101	26108491	6756615	4107644	178721852	581419167	781776300	1676363364	-0.22%
L1_IOM110	26926311	6637828	4016483	182711443	537565610	769580491	1671778241	7.34%
L1_IOM111	26918681	6596263	3993383	181463088	524479564	765915675	1672292207	9.60%
L2_IOM100	24829979	6480055	3798617	167329735	569617045	749078537	1673880756	1.82%
L2_IOM101	25186153	6674722	3746167	169056524	567717723	758147334	1674288189	2.14%
L2_IOM110	25023492	6626141	3554062	181981151	544640816	767793792	1672052693	6.12%
L2_IOM111	24901785	6527208	3465305	182706363	541850677	768954693	1672350163	6.60%
L3_IOM100	24752809	6496051	3798298	172086843	580541427	760709680	1673698605	-0.07%
L3_IOM101	25234489	6634956	3732110	176093727	590075549	775351342	1674265847	-1.71%
L3_IOM110	24992097	6526048	3537258	185428001	544634967	776220778	1671973193	6.12%
L3_IOM111	24918077	6503766	3443714	179482937	527423338	761075054	1672124889	9.09%
NTA_IOM100	25833397	6779981	4144530	172551482	581159757	761845277	1674851900	-0.17%
NTA_IOM101	26253504	6881928	4143657	170041328	577361946	760554960	1675706124	0.48%
NTA_IOM110	26562484	9017609	4187411	181859900	544242122	767497646	1671876087	6.19%
NTA_IOM111	26730900	9013035	4172751	180662672	522934693	763959197	1672349983	9.86%
No Prefetch	25018068	6662494	4191812	161521262	580158527	748723369	1673709476	0.00%

Appendix C.5

Dataset for Uniform Distribution with Compiler prefetching off, run with 4 MPI tasks, 4 OpenMP threads/task with no coprocessor offloading

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	40413648	12959641	6937841	256038503	828071103	1154438745	2594605491	3.89%
L1_IOM101	41044428	13577617	6977240	258370630	822131540	1167388278	2600547597	4.58%
L1_IOM110	42718631	13250215	6858948	272939838	770931280	1166304152	2603071501	10.52%
L1_IOM111	42304172	12866936	6686728	276120928	770829800	1175018081	2574370551	10.53%
L2_IOM100	39148419	12873718	6479170	257089936	834141983	1157004153	2591241667	3.18%
L2_IOM101	40118267	13427611	6316641	258068737	821182869	1167509801	2597054695	4.69%
L2_IOM110	39408805	12981382	6296023	275286148	794805420	1176354964	2602242616	7.75%
L2_IOM111	39850010	13102308	6006727	277177804	791047907	1177742881	2605178036	8.18%
L3_IOM100	39138143	12934766	6472509	255745956	831186364	1154839756	2592002709	3.52%
L3_IOM101	40107565	13446408	6283942	258332267	822954890	1167889682	2607660287	4.48%
L3_IOM110	39835096	13674395	6378851	275010613	789903764	1175853736	2607892403	8.32%
L3_IOM111	39736447	13668126	6175728	273118861	776795429	1165050981	2608185235	9.84%
NTA_IOM100	40631275	13152087	6969650	255478535	824474602	1153586709	2589159477	4.30%
NTA_IOM101	41878004	13791031	6993544	258331636	822783422	1168759699	2608589407	4.50%
NTA_IOM110	42080198	16695752	7066892	275393993	779644684	1176393462	2603138943	9.51%
NTA_IOM111	41592419	16566806	6973256	274638635	762068794	1170846054	2594194123	11.55%
No Prefetch	39557837	12838951	7023312	243015330	861552943	1144450218	2601710639	0.00%

Appendix C.6

Dataset for Uniform Distribution with Compiler prefetching on, run with 4 MPI tasks, 4 OpenMP threads/task with no coprocessor offloading

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	40592014	13223226	7005302	256137514	825329581	1155872456	2609423560	3.62%
L1_IOM101	41430998	13149098	6939471	255260323	807386664	1155812470	2610249463	5.71%
L1_IOM110	42684168	13866352	6923727	273098486	763337044	1168239147	2607331207	10.86%
L1_IOM111	42336092	13431104	6826895	275481991	765418328	1175274680	2602637943	10.61%
L2_IOM100	39374639	12994468	6541648	255042360	823979100	1150875697	2605648225	3.77%
L2_IOM101	39858747	13076902	6271759	256553596	816563058	1162123458	2592344832	4.64%
L2_IOM110	39530424	13518099	6321919	273252525	788341661	1169214588	2602261332	7.94%
L2_IOM111	39801616	13554874	6091890	276509641	785329449	1177622121	2592707549	8.29%
L3_IOM100	39645328	12945886	6496673	254815217	828714417	1148335425	2611839684	3.22%
L3_IOM101	39963109	13395545	6314529	257921463	821044614	1167860567	2607955225	4.12%
L3_IOM110	39650791	13225170	6268943	272613091	788136555	1166577523	2604810332	7.96%
L3_IOM111	40057694	13567664	6060131	277086278	792211088	1177672614	2610806992	7.48%
NTA_IOM100	41002223	13370847	6989443	256470448	828540243	1155939737	2583176451	3.24%
NTA_IOM101	41891211	13732572	7038479	256488133	812495344	1161411405	2612779735	5.12%
NTA_IOM110	42656882	16891206	7084274	275548042	779148500	1175584221	2608618777	9.01%
NTA_IOM111	42338598	16749848	6998449	276083235	764570545	1177586697	2592388864	10.71%
No Prefetch	39487872	13133504	7096340	242857608	856304163	1144838338	2594081649	0.00%

Appendix C.7

Dataset for Non-uniform Distribution with Compiler prefetching off, run with 4 MPI tasks, 4 OpenMP threads/task with no coprocessor offloading

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	136742323	33182934	17453713	900829329	2607237244	4068449033	9132952257	3.16%
L1_IOM101	140892904	33001652	17409747	902584730	2586955006	4090006257	9233027981	3.91%
L1_IOM110	146081615	33935377	17133867	965057306	2476054782	4118696686	9187499971	8.03%
L1_IOM111	146487566	33426033	16881852	966748264	2452162642	4113607528	9198608487	8.92%
L2_IOM100	132845537	33019001	15710984	898676476	2604391502	4058671365	9127170548	3.26%
L2_IOM101	135801131	33451639	15508242	907315851	2598818781	4117667296	9199834293	3.47%
L2_IOM110	134444718	32911622	14478233	964959774	2522211248	4121235227	9154124203	6.31%
L2_IOM111	133843678	33309898	14474301	969112551	2509300542	4119750471	9188936544	6.79%
L3_IOM100	133803235	33611965	16125927	896882405	2619871098	4043360654	9249352580	2.69%
L3_IOM101	134981155	33262712	15461551	907137217	2589067698	4117149427	9198241295	3.83%
L3_IOM110	134879150	32457943	14540733	968870685	2519774272	4132282656	9214731261	6.40%
L3_IOM111	133853479	33226375	14390624	968654268	2500372906	4118262479	9188828560	7.13%
NTA_IOM100	140158160	34590671	17777343	900481831	2605647314	4062911450	9203819175	3.21%
NTA_IOM101	142208296	34862365	17723845	903474973	2600025509	4097690738	9239530813	3.42%
NTA_IOM110	145346099	48348379	18098190	968677038	2459153446	4136177747	9240871695	8.66%
NTA_IOM111	144951488	47492704	17936533	968555637	2450824145	4126472550	9190070904	8.97%
No Prefetch	133584719	33394209	18000462	845271691	2692200736	3990041447	9160682940	0.00%

Appendix C.8

Dataset for Non-uniform Distribution with Compiler prefetching on, run with 4 MPI task, 4 OpenMP threads/task with no coprocessor offloading

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	139102380	32991618	17466174	896085289	2600029065	4045196763	9240316973	3.21%
L1_IOM101	141113789	34182826	17569009	899676392	2585300081	4071997286	9207186393	3.76%
L1_IOM110	145711282	34029917	17106952	960130803	2458011688	4094977516	9107587184	8.50%
L1_IOM111	145585659	33465318	16664266	967754185	2443226047	4116069702	9173652789	9.05%
L2_IOM100	132814544	32600960	15627233	898961977	2604692366	4060644123	9127783387	3.03%
L2_IOM101	134096909	33876921	15427341	907341593	2588299467	4117788525	9128517277	3.64%
L2_IOM110	134226795	32582877	14327997	969673602	2519532033	4138949567	9154041365	6.20%
L2_IOM111	134064121	33327584	14395165	970806473	2503456685	4133640109	9224401135	6.80%
L3_IOM100	133408545	32163065	15472408	902968743	2604080247	4077933298	9196714156	3.06%
L3_IOM101	134872074	32887518	15418920	906052996	2613487092	4110952004	9153165333	2.71%
L3_IOM110	135116104	33201042	14607751	968300898	2529253910	4133499610	9143806371	5.84%
L3_IOM111	134760389	34003279	14446783	969903334	2517189108	4121319206	9199250535	6.29%
NTA_IOM100	139917466	34358667	17795058	900438472	2612387307	4061015422	9201559579	2.75%
NTA_IOM101	140668618	35885907	17827760	903633655	2577576503	4101691108	9134972491	4.04%
NTA_IOM110	145021439	47825906	17997150	961296556	2457885961	4098219575	9153835737	8.50%
NTA_IOM111	144678315	48076342	18028620	969828572	2460558988	4127841361	9164135793	8.40%
No Prefetch	133077245	33241931	17921641	848032189	2686209580	4006220922	9150751817	0.00%

Appendix C.9

Dataset for Uniform Distribution with Compiler prefetching off, run with 16 MPI task, 1 OpenMP thread/task with no coprocessor offloading

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	39869919	12729473	6682434	262996568	780420295	1188544072	9240316973	4.66%
L1_IOM101	39942144	12841108	6657561	272251659	788755148	1235483798	9207186393	3.64%
L1_IOM110	43516437	14238426	6776488	252706990	676736573	1081855400	9107587184	17.33%
L1_IOM111	44090028	13590464	6578830	289449926	726061328	1235021668	9173652789	11.30%
L2_IOM100	40820126	12988096	6097767	259834677	784199460	1174164627	9127783387	4.20%
L2_IOM101	38881862	12506508	6107691	265352266	778057876	1203926481	9128517277	4.95%
L2_IOM110	40985722	14388639	6356384	281316264	746167243	1203602309	9154041365	8.85%
L2_IOM111	38881730	13228516	6062462	293649226	757623671	1252828864	9224401135	7.45%
L3_IOM100	40105145	13195797	6157674	270184308	804639318	1221086985	9196714156	1.70%
L3_IOM101	39184672	13025942	6193066	273977628	792194888	1243298378	9153165333	3.22%
L3_IOM110	40965871	15310930	6612941	287690244	762975476	1230687531	9143806371	6.79%
L3_IOM111	41381375	14223983	6000192	269237742	715500402	1149029721	9199250535	12.59%
NTA_IOM100	39739081	12818618	6739319	263859609	785157955	1192416155	9201559579	4.08%
NTA_IOM101	41669843	13667640	6757732	269664931	796414733	1223621159	9134972491	2.71%
NTA_IOM110	43121389	18012825	6981495	290487836	735356392	1242629386	9153835737	10.17%
NTA_IOM111	42898130	17399903	6829983	287584495	727481281	1227031194	9164135793	11.13%
No Prefetch	39004274	12760305	6764767	249267780	818590707	1178036946	9150751817	0.00%

Appendix C.10

Dataset for Uniform Distribution with Compiler prefetching on, run with 16 MPI task, 1 OpenMP threads/task with no coprocessor offloading

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	41558116	13680827	6906309	270184044	799506258	1221051762	2720265785	-2.75%
L1_IOM101	43229496	14280956	6816259	263339588	777568677	1194727395	2767840765	0.07%
L1_IOM110	43293598	13217451	6566910	276650348	720362061	1183757874	2680590291	7.42%
L1_IOM111	40591495	12501069	6369190	291781329	724550227	1244926245	2543138867	6.88%
L2_IOM100	41145338	14290980	6358006	268027886	803117329	1211265944	2767855559	-3.22%
L2_IOM101	37275219	12285955	5942915	266214603	782626923	1207833775	2451546513	-0.58%
L2_IOM110	41379818	13773184	6063793	268410785	726324415	1148717317	2760582845	6.65%
L2_IOM111	38919174	13211486	6028191	263329837	705618299	1123911276	2595300276	9.31%
L3_IOM100	39769214	13168469	6229848	254803443	783115452	1151354630	2669869319	-0.65%
L3_IOM101	39532032	13980151	6387170	242928040	742796331	1101405002	2617506424	4.53%
L3_IOM110	40547212	13402852	6058748	282248349	746999163	1207606330	2688577644	3.99%
L3_IOM111	39448272	13206565	5955230	290383156	753926871	1238990548	2622419208	3.10%
NTA_IOM100	40465058	13477006	6825794	243161069	750622889	1098589110	2615047437	3.53%
NTA_IOM101	43835122	14544434	6816377	267508477	793244770	1213747332	2766502955	-1.95%
NTA_IOM110	43160028	17220040	6817357	279449282	729227175	1195664457	2674432057	6.28%
NTA_IOM111	40844676	16418801	6786776	285717850	720189404	1219163840	2541440889	7.44%
No Prefetch	39807132	13673038	6936308	227946765	778079949	1076322134	2660856783	0.00%

Appendix C.11

Dataset for Non-uniform Distribution with Compiler prefetching off, run with 16 MPI task, 1 OpenMP threads/task with no coprocessor offloading

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	137709486	39115581	17919249	886090280	2447734995	4006154416	9250874856	4.93%
L1_IOM101	139682524	39172347	17736453	915756167	2453215485	4161657728	9254352151	4.72%
L1_IOM110	144913198	39223591	17709796	954980966	2306941143	4080234259	9199556863	10.40%
L1_IOM111	146280627	39260638	17371145	963466992	2267227094	4107589474	9289662781	11.95%
L2_IOM100	133835243	37987716	16369905	899030010	2482538144	4064745507	9162810275	3.58%
L2_IOM101	135060569	39094659	16088407	903968523	2454004520	4107777526	9248781256	4.69%
L2_IOM110	133256836	39376756	15562161	958083566	2375136413	4093462487	9165398835	7.75%
L2_IOM111	133891694	39018354	15041096	968507099	2365039945	4128696412	9149485128	8.15%
L3_IOM100	134411804	37847874	16121799	895003658	2494890141	4046652619	9360070487	3.10%
L3_IOM101	134848115	39092997	15921175	902100641	2483171305	4099200240	9238953833	3.56%
L3_IOM110	133894608	39454579	15632558	959484870	2376823089	4098745071	9201218319	7.69%
L3_IOM111	133442686	39439752	15035666	984399889	2405799508	4196189070	9125137509	6.56%
NTA_IOM100	138789039	39703705	17961652	904922661	2507980368	4091580221	9285939559	2.59%
NTA_IOM101	141111865	40805627	17844744	912449546	2440409433	4146515507	9232910988	5.22%
NTA_IOM110	143443126	53358911	18035640	959326483	2282875881	4098083538	9181339968	11.34%
NTA_IOM111	144227427	53440013	18159488	968658505	2286068814	4129357892	9226750011	11.21%
No Prefetch	133656337	39080029	18453594	847889721	2574794528	4016065272	9228734705	0.00%

Appendix C.12

Dataset for Non-uniform Distribution with Compiler prefetching on, run with 16 MPI task, 16 OpenMP thread/task with no coprocessor offloading

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	138281481	38369291	17801630	901758389	2469273119	4077047978	9279885808	5.04%
L1_IOM101	140679955	38956781	17625757	889597640	2429393288	4041987846	9341129772	6.57%
L1_IOM110	143849606	39384244	17501171	967317231	2313591094	4132534810	9124116127	11.02%
L1_IOM111	145382206	39054871	17235664	971006174	2296131604	4139988752	9178287924	11.70%
L2_IOM100	132628220	38340678	16457407	890404517	2499694563	4025902132	9103284312	3.87%
L2_IOM101	133985169	39122913	15911338	894336794	2477001727	4063871068	9222118455	4.74%
L2_IOM110	133997523	39877164	15555266	965057316	2377432739	4122571409	9205415983	8.57%
L2_IOM111	134539165	38302429	14987991	977617384	2388893188	4167713543	9195133411	8.13%
L3_IOM100	133476461	37911602	16170588	894286636	2503980170	4043590391	9238485491	3.70%
L3_IOM101	135327119	39013179	16104361	894049378	2468614226	4062492255	9215075291	5.06%
L3_IOM110	133610167	38216818	15095218	953002390	2364781199	4072295453	9212482756	9.05%
L3_IOM111	134080752	38922296	15327733	963625180	2376956848	4108251024	9302576211	8.59%
NTA_IOM100	138309697	39167407	18109919	907510654	2474802184	4103137354	9197377509	4.82%
NTA_IOM101	141069149	41063807	17957127	889741725	2442371445	4042644957	9233834007	6.07%
NTA_IOM110	144503733	54070574	18120126	959328104	2283756840	4098083539	9230272004	12.17%
NTA_IOM111	141911255	53710411	18078251	973666514	2314909771	4151201692	9122294488	10.97%
No Prefetch	133198815	37516235	18027918	854412648	2600228519	4047112374	9201101156	0.00%

Appendix C.13

Dataset for Uniform Distribution with Compiler prefetching off, run with 1 MPI task, 16 OpenMP threads/task with coprocessor offloading on

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	39326057	11384210	6580143	249223404	870450169	1118421734	2508404119	2.66%
L1_IOM101	40182507	11641990	6531039	250611427	865500577	1129067829	2510569549	3.21%
L1_IOM110	41305771	11511111	6467555	267154769	813513471	1136806505	2503681639	9.02%
L1_IOM111	41472107	11491061	6378786	267073587	808726915	1135687693	2504257835	9.56%
L2_IOM100	38321199	11440606	6156133	249446833	872397249	1118967329	2506740804	2.44%
L2_IOM101	38547149	11512445	5973964	251360853	866834890	1130900178	2507015220	3.06%
L2_IOM110	38452562	11510531	5955610	265710090	825470097	1133267458	2504092264	7.69%
L2_IOM111	38465018	11516421	5749134	267270900	823003772	1136168080	2504267311	7.96%
L3_IOM100	38188928	11326024	6129539	248638798	870717836	1116991471	2506755848	2.63%
L3_IOM101	38817903	11604762	5993919	251285774	865200227	1130715730	2507798987	3.24%
L3_IOM110	38286251	11524948	5962121	266751505	827762715	1135818958	2504162864	7.43%
L3_IOM111	38408691	11495668	5749968	268369136	822946564	1138851481	2504203925	7.97%
NTA_IOM100	39788942	11876217	6649109	248415916	870830833	1116446967	2509354321	2.61%
NTA_IOM101	40214985	12058515	6612664	251797585	867296145	1131967269	2509222733	3.01%
NTA_IOM110	40954073	15213541	6665919	266696800	817165217	1135682922	2503781473	8.62%
NTA_IOM111	40915970	15288667	6657971	268425221	809265066	1138991793	2504516683	9.50%
No Prefetch	38229381	11562147	6692965	235803110	894213325	1106211009	2506115931	0.00%

Appendix C.14

Dataset for Uniform Distribution with Compiler prefetching on, run with 1 MPI task, 16 OpenMP threads/task with coprocessor offloading on

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	39347543	11492752	6588420	249795688	872957902	1119819957	2508973924	2.90%
L1_IOM101	40072417	11634519	6536480	251067480	865795490	1130179658	2509970717	3.70%
L1_IOM110	41340158	11530726	6455411	266295401	813663817	1134704512	2503723004	9.50%
L1_IOM111	41330702	11530405	6405914	268504991	806967848	1139187783	2504249499	10.24%
L2_IOM100	38252031	11411494	6144504	249404789	871871483	1118865057	2506801420	3.02%
L2_IOM101	38875972	11564824	5988105	250726084	865754737	1129346971	2507428744	3.70%
L2_IOM110	38444930	11553492	5945828	266728294	830842283	1135761840	2503965295	7.58%
L2_IOM111	38366237	11521293	5728955	268279319	821663403	1138635134	2504477848	8.61%
L3_IOM100	38080652	11322326	6143705	249028841	870607904	1117947219	2506586296	3.16%
L3_IOM101	38903531	11559014	5954816	251224756	867115921	1130567428	2507348415	3.55%
L3_IOM110	38442647	11526112	5931299	266568646	828467887	1135372062	2503916736	7.85%
L3_IOM111	38363049	11465487	5717646	267803640	822120071	1137471549	2504308861	8.56%
NTA_IOM100	39767711	11868170	6634840	249299619	871732007	1118607297	2508707013	3.04%
NTA_IOM101	40297857	12042923	6606708	252482684	870604436	1133641785	2509231735	3.16%
NTA_IOM110	41019266	15204088	6688119	267119414	814345490	1136720213	2503724157	9.42%
NTA_IOM111	40993561	15253685	6661047	268203697	808450165	1138451573	2504599809	10.08%
No Prefetch	38359035	11565603	6704552	236896663	899033490	1108884665	2506114604	0.00%

Appendix C.15

Dataset for Non-uniform Distribution with Compiler prefetching off, run with 1 MPI task, 16 OpenMP threads/task with coprocessor offloading on

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	25844361	6570189	4084231	170349264	569310928	756460916	1675037744	2.12%
L1_IOM101	26257830	6733252	4101129	169772662	569989899	759898409	1675866285	2.00%
L1_IOM110	27054512	6679034	4010244	178034833	526162707	758146494	1671833147	9.54%
L1_IOM111	27160469	6654830	3989767	178007002	525968804	757466141	1672237581	9.57%
L2_IOM100	24959563	6484565	3772862	168510237	567457437	751966481	1673774232	2.44%
L2_IOM101	25229408	6615626	3725066	171302255	580449652	763637551	1673977403	0.21%
L2_IOM110	25128135	6580242	3512271	177918480	529019240	757860920	1672047327	9.05%
L2_IOM111	25280426	6562229	3418126	178077222	530701060	757636118	1672280321	8.76%
L3_IOM100	25124110	6512749	3765702	168364989	577516425	751608021	1674109784	0.71%
L3_IOM101	25473340	6627203	3718980	172095853	567070481	765578797	1674448609	2.51%
L3_IOM110	25188609	6530186	3525213	178546233	539134956	759397227	1672088945	7.31%
L3_IOM111	25120078	6546665	3450671	181547423	531184698	766121781	1672251263	8.68%
NTA_IOM100	26107652	6811373	4107766	171203880	580574981	758550891	1674974821	0.18%
NTA_IOM101	26334314	6918521	4122845	169650871	567215305	759600585	1675494541	2.48%
NTA_IOM110	26771945	9035574	4154988	179985691	531632753	762916916	1671908201	8.60%
NTA_IOM111	26870187	9107617	4168875	183494896	528823809	770882656	1672328625	9.08%
No Prefetch	25312472	6658391	4144493	157484659	581649228	738831964	1673410161	0.00%

Appendix C.16

Dataset for Non-uniform Distribution with Compiler prefetching on, run with 1 MPI task, 16 OpenMP threads/task with coprocessor offloading on

Numbers are trimmed averages over 5 runs. Numbers are per thread.

IOM100 - Prefetching IN array only, IOM101 - Prefetching IN, M arrays,

IOM110 - Prefetching IN, OUT arrays, IOM111 - Prefetching IN, OUT, M arrays

	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_L3_TCM	PAPI_LD_INS	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_DP_OPS	Prefetch gain
L1_IOM100	25755225	6599969	4091290	166722114	566738786	747594764	1675148177	2.21%
L1_IOM101	26380303	6774305	4068216	166685900	566669405	752351974	1676063817	2.22%
L1_IOM110	27150004	6656631	3992091	180053598	529775230	763083029	1671839107	8.59%
L1_IOM111	27492895	6707476	3951116	183542920	522980038	770999745	1672293120	9.76%
L2_IOM100	24988559	6497612	3779750	168949846	568893065	753039937	1673503308	1.84%
L2_IOM101	25280140	6638207	3728487	170491689	569328925	761654749	1674190417	1.77%
L2_IOM110	25106749	6529255	3530695	180737100	541889106	764739463	1672099813	6.50%
L2_IOM111	25061896	6553661	3441206	184231101	535481411	772682889	1672310120	7.61%
L3_IOM100	24960462	6489325	3766112	172331483	581155112	761308085	1673892111	-0.27%
L3_IOM101	25337825	6639032	3705833	176911242	571147202	777351288	1674086439	1.45%
L3_IOM110	25171973	6492936	3535261	180150488	533073555	763315499	1672066081	8.02%
L3_IOM111	25079870	6544388	3458265	180300029	527217520	763071430	1672200183	9.03%
NTA_IOM100	25933078	6797736	4125741	166890273	570478481	748006360	1674844933	1.57%
NTA_IOM101	26381635	6991657	4116820	169992098	568829708	760435292	1675393131	1.85%
NTA_IOM110	26796349	8995639	4156893	178564316	533495562	759442075	1672001267	7.95%
NTA_IOM111	27018264	9137754	4166579	183877978	528143228	771821006	1672321788	8.87%
No Prefetch	25231926	6659171	4148647	157637066	579562037	739228355	1673221889	0.00%

References

1. Aashish Phansalkar , Ajay Joshi , Lizy K. John, “Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite”, Proceedings of the 34th annual international symposium on Computer architecture, June 09-13, 2007, San Diego, California, USA, doi>10.1145/1250662.1250713
2. John D. McCalpin, IBM, Keynote address at the Third Annual IEEE Workshop on Workload Characterization, Austin, Tx, 2000 www.cs.virginia.edu/~mccalpin/DOE/
3. Browne, S., Deane, C., Ho, G., Mucci, P. “PAPI: A Portable Interface to Hardware Performance Counters”, Proceedings of Department of Defense HPCMP Users Group Conference, June, 1999, doi>10.1.1.117.6801
4. Wilhelm Anacker and Chu Ping Wang. "Performance Evaluation of Computing Systems with Memory Hierarchies", IEEE Transactions on Electronic Computers, Dec.1967, pp. 764-773, doi>10.1109/PGEC.1967.264722
5. Tien-Fu Chen and Jean-Loup Baer. "Effective hardware-based data prefetching for high performance processors", IEEE Transactions on Computers, May 1995, pp. 609-623, doi>10.1109/12.381947
6. Fredrik Dahlgren, Michel Dubois, and Per Stenstr om. "Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors", International Conference on Parallel Processing, ICPP, Vol. 1. Aug. 1993, pp. 5-63, doi>10.1109/ICPP.1993.92
7. J. D. Gindele. "Buffer Block Prefetching Method", IBM Technical Disclosure Bulletin, July 1977, pp. 696-697
8. G Hinton et al. "The Microarchitecture of the Pentium 4 Processor", Intel Tech Journal 5, Feb 2001. URL:www.intel.com/technology/itj/archive/2001.htm
9. Daniel Gracia Perez, Gilles Mouchard, and Olivier Temam. "MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms", Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2004, pp. 43-54, doi>10.1109/MICRO.2004.25
10. Alan J. Smith. "Sequential Program Prefetching in Memory Hierarchies", IEEE Transactions on Computers, Dec. 1978, pp. 7-21, doi>10.1109/C-M.1978.218016
11. Alan Jay Smith. "Cache Memories", ACM Computing Surveys 14.3, Sept. 1982, pp. 473-530, doi>10.1145/356887.356892
12. Krishnaiyer et. al., “Compiler-based Data Prefetching and Streaming Non-temporal Store Generation for the Intel Xeon Phi Coprocessor”, Workshop on Multithreaded Architectures and Applications , MTAAP 2013

13. Callahan et. al., "Software Prefetching", Proceedings of the fourth international conference on Architectural support for programming languages and operating systems (ASPLOS IV), 1991, pp. 40-52, doi>10.1145/106973.106979

14. Gornish et. al., "An integrated hardware/software data prefetching scheme for shared-memory multiprocessors", International Conference on Parallel processing, ICPP 1994, Vol. 2, pp-281-294, doi> 10.1109/ICPP.1994.57

15. Parthasarthy et. al, "The interaction of software prefetching with ILP processors in shared-memory systems", Proceedings of the 24th annual international symposium on Computer architecture, ISCA 1997, pp. 144-156, doi>10.1145/384286.264158

16. Badawy et. al., "The Efficacy of Software Prefetching and Locality Optimizations on Future Memory Systems", Journal of Instruction Level Parallelism, 2004, doi>10.1.1.73.8314

17. Malhotra et. al., “A distributed memory fast multipole method for volume potentials”, The International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 13, Nov 16-22, 2013.